Using Layered Bottlenecks for Virtual Machine Provisioning in the Clouds

Yasir Shoaib, Olivia Das

Electrical and Computer Engineering Department, Ryerson University, Toronto, ON
Email: yasir.shoaib@ryerson.ca, odas@ee.ryerson.ca

Abstract—Meeting the QoS objectives of fluctuating web workload requires techniques built on performance models, controller algorithms, monitors, etc. To meet the demands, we propose a controller algorithm using performance models that addresses the dynamic provisioning problem of multi-tier web applications in the cloud computing domain through addition of resources. The proposed algorithm aims to attain response time objectives by identifying “layered bottlenecks” and on this basis adding virtual machines (VM) and virtual CPUs, while keeping a check on limits such as spare VMs, processors-per-VM and replicas-per-VM. Here, Layered Queueing Network (LQN) performance models are used, alongside jLQNInterface, a tool developed in Java that allows solving, analyzing, and manipulating LQN models through the implemented API. The algorithm has been implemented using the tool and its applicability is demonstrated through a case study. By comparing two cases, it is shown that the proposed algorithm by using layered bottlenecks results in a model that satisfies the objectives with fewer resources.

Keywords—Cloud computing, dynamic provisioning, performance modeling, layered bottlenecks

I. INTRODUCTION

Application providers (AP) strive for deploying web applications that meet high performance standards even when workload demands are at their peak. They meet these request demands either through the management of their own web-server farms or through the purchase of hardware and software as services from cloud computing service providers.

The selling point of cloud computing is the multi-dimensional ease with which computing resources could be borrowed and utilized through the Internet. The resources, viz. processors, network, software, etc., remain in cloud provider’s data centers and can be added and removed as needed by the customer [1], [2]. With cloud computing’s utility-based pricing model, the customers pay for the resources used; in comparison, they would have incurred capital expenses if resources were purchased [2]. These customers considering their spatial, temporal and monetary constraints, could thereby rely on services the “cloud” [3] is known to offer. The well-known cloud services offered by providers such as Amazon, Google, Microsoft, etc. are: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS) [3], [4].

Cloud computing, however, faces many technical challenges, one of which deals with autonomously provisioning adequate resources, i.e. dynamically adding and removing resources — including virtual machines (VMs) — to handle the fluctuating Internet user request demands [3], [4]. Under-provisioning of resources cause web application end-users to experience excessive delays, especially during demand surges. Eventually, due to poor performance, disgruntled users leave the site, incurring loss to the AP businesses — as seen happening with eCommerce sites [5]. On the other hand, over-provisioning leads to higher costs for the cloud provider (CP) due to management of large number of servers that because of being under-utilized cause excess power consumption and heat dissipation in their data centres. The loss is not only restricted to CP but also extends to AP, who have constrained budgets but have to pay for unnecessary VM instances running. The quintessential scenario would be the dynamic provisioning of resources following the fluctuating demands to meet various Quality-of-Service (QoS) requirements and reduction of cost and power, which in practice is quite a challenging endeavor [3].

Our proposed research aims to solve the dynamic VM provisioning problem by adding resources through a controller algorithm that relies on performance models for decision making. Calheiros et al. [6] have looked into the VM provisioning in clouds by using their provisioning algorithm and a simple single-tier queueing network performance model for PaaS and SaaS providers. In our research, we intend to explore provisioning for multi-tier applications and shared resources through the use of Layered Queueing Networks (LQN) performance models, which are unfettered by shortcoming of queueing networks (QN) and are ideal for modeling software systems [7].

One way to provision is by identifying bottlenecks through resource utilizations. Bottlenecks would be resources that exceed a set threshold utilization. Once the bottleneck is found, instances of VMs or associated virtual processors are added to improve performance. However, applications running on the VM could also be bottlenecks instead of the virtual processor. In such scenarios it is essential to distinguish if the virtual CPU or the software is limiting the performance and as an effect simplifying to whether virtual CPUs should be increased or the VM instances. To this end, we use the notion of “layered bottlenecks” [8] that rely on the “BStrength” [8] metric to identify bottlenecks associated with software processes. An algorithm that bases the provisioning decisions on virtual processor utilizations and on BStrength of processes is presented. We also introduce jLQNInterface, a tool that allows solving, analyzing,
and manipulating LQN models through its API, which has been used for implementation of the controller algorithm.

In this paper, we focus on VM provisioning by AP who are IaaS customers and who in-turn serve SaaS to end-users. The main contributions are VM provisioning using:

1) jLQNInterface to solve, analyze and manipulate LQN models of multi-tiered applications.
2) Provisioning controller algorithm based on “layered bottlenecks” [8] by:
   a) Adding VMs
   b) Increasing number of virtual processors
   c) Considering constraints such as:
      i) Maximum available spare VMs
      ii) Maximum processors-per-VM
      iii) Maximum replication-per-VM
3) Case study demonstrating applicability of the algorithm.

Following is an outline of the paper. Section II lists our general assumptions related to clouds. Section III mentions the related works. Section IV provides a short summary of LQN models, LQX system, jLQNInterface and layered bottlenecks. Section V explains the controller algorithm and a provisioning framework using the controller. Section VI describes the case study and results. Section VII presents the conclusions.

II. ASSUMPTIONS

We distinguish between CP, AP and end-users [9]. Figure 1 shows the cloud control hierarchy. CP provide IaaS, which includes hardware and virtualized resources such as VMs that run on the hardware. Its customers directly access the virtual resources only. AP are IaaS customers, who can create and instantiate VMs and install their choice of operating environment, which includes Operating Systems and software development kits (SDK). Furthermore, these VMs may be allocated and deallocated on-demand through facilities — such as API — provided by the CP, allowing resource elasticity [1], [3]. Although, it is possible that AP be a PaaS customer only, instead of being IaaS customer, however, for simplicity we assume that AP is an IaaS customer with ability to create VMs and assign virtual processors to the VMs. AP develop their web applications and deploy them on VMs, thereby providing SaaS to their customers: end-users. The provisioning decisions made by AP are to accommodate the workload demands of the applications that are accessed by the end-users.

In this paper we assume that provisioning is done by AP, following the same argument as provided by Calheiros et al. [6]. CP are not aware of the specific demands of the hosted applications. If performance of the software is to be predicted, a performance model would need to exist, which the AP can develop. Therefore, with respect to VM provisioning, we consider that the decision to add or remove VM instances is up to the AP.

III. BACKGROUND

“Cloud provisioning” [4] [6] involves resource allocation — and administration — to facilitate the deployment of applications that would happen to run in the cloud domain. Calheiros et al. [6] list following as three steps to cloud provisioning: 1) VM provisioning: creation of VMs, 2) Resource provisioning: association of created VMs to adequate hardware resources, 3) Application provisioning: load balancing of incoming user requests to appropriate VMs. In this paper our focus is towards VM provisioning. Here, we use the term “resource provisioning” generally, referring simply to addition of resources.

A. Related Works

Previous works have dedicatedly looked into resource provisioning and these efforts have been extended to find their application in the cloud as well. The important works below highlight the approaches that have been adopted.

Menasce [10] presents a QoS controller design that relies on monitored data and QN models to derive an optimal configuration (e.g. thread count and maximum queue size) meeting the objectives of response time, throughput and rejection probability for a multi-tiered electronic commerce website. The controller uses hill-climbing algorithm with objective to maximize an aggregate QoS metric, thereby meeting the desired goals.

Urgaonkar et al. [11] propose a combination of workload prediction, virtualized technology, admission control, proactive and reactive dynamic provisioning to meet given response time deadlines of a multi-tier system. The QN model receives inputs from the workload prediction to determine how many servers to provision. Use of VMs help in quick adjustment of resources.

Li et al. [9] use LQN models in maximizing the profit of the cloud while considering QoS (response time and throughput in their case) as constraints. The objectives are met by optimally placing VMs on physical machines. They use NFM to solve the deployment optimization problem.
Work by Zheng [12], Huber et al. [13], Calheiros et al. [6], Ruiqing et al. [14] are the most similar to our work. Below we highlight the similarities and differences.

Zheng [12] presents a framework that automatically allocates server resources when response time objectives are violated. Their controller uses hill-climbing algorithm, which begins at an initial system state and evaluates the neighbor states, choosing the neighbor that results in the lowest cost. In a similar pattern, the algorithm continues searching through the neighbors until the optimal cost is found. Through a few simulation experiments and a case study, the workings of the provisioning framework and the controller are demonstrated. In comparison, our approach does not evaluate the neighbor states for decision making but instead uses bottleneck information to traverse through the neighbors and meet the objectives.

Huber et al. [13] present a framework allowing dynamic allocation and de-allocation of VMs and virtual CPUs, based on the run-time demands of the virtualized systems. The framework comprises of a “control loop” [13] and uses performance models. A case study using a SPECjEnterprise2010 benchmark deployment was conducted for three scenarios where a new service was added, workload was increased and decreased. The evaluation showed promising results with meeting response time guarantees by dynamically managing resources. For an example week workload, the paper showed maximum savings of 40% of resources may be achieved when using the dynamic approach in comparison to a static allocation that would meet QoS guarantees. In comparison to above approach, we employ layered bottlenecks to guide our search.

Apart from previous works, Calheiros et al. [6] present a unique perspective where resource provisioning is done by SaaS and PaaS providers, who are described to have a clear view from the application perspective and are in a better state to make VM provisioning decisions. The proposed algorithm by Calheiros et al. [6] aims to satisfy QoS requirements such as response time, utilization and rejection rate of VMs based on negotiated QoS attributes. The solution employs workload prediction and performance modeling and uses VM monitoring. A simple QN model of the system is depicted. Furthermore, admission control is adopted by determining the maximum queue size of a queueing station and rejecting requests that arrive if the queue is full. The simulation results show promising results where the VM hours were reduced with no or about none rejection rate and the negotiated response times were met. Our approach closely relates to their perspective and approach, however, our solution is aimed towards multi-tiered web applications, where each application may span multiple VMs.

Ruiqing et al. [14] provide a heuristic search algorithm to find optimal deployment configuration for a multi-tiered web application in the clouds. A given configuration includes the number of VMs allocated to each tier. The main features of the algorithm include a utility function that adds another level of QoS along with response time, a rule-set database to find initial configuration for a given workload, and a pruning algorithm which finds the optimal deployment. The pruning algorithm adds/removes VMs of tiers of an application and uses differences in workloads and the utility function to reach optimal solution quicker. Alongside, LQN models are used for decision making. They evaluate their approach with a two-tier web application. The main difference between their work and ours is for finding optimal configuration they use an expert system — relying on and updating the rule-set database — and their pruning algorithm, whereas we focus directly on layered bottlenecks.

IV. PERFORMANCE MODELING

In this section a brief description of LQN modeling, LQX system, jLQNInterface and layered bottlenecks are provided.

A. Layered Queueing Networks (LQN)

Layered Queueing Networks (LQN) [15] models are based on extended QN and have been used widely in many performance studies (e.g. [7], [16]). Modeled entities include hardware resources as processors, software processes as tasks, service classes as entries, and finer operation steps of software as activities. Entities have associated multiplicity, i.e. multiple entities with one queue. Processors — and thereby associated resources — can be replicated with each copy having their own queue. These models are able to depict nested interactions between servers considering simultaneous resource possession, and provide results such as response time, throughput and utilization of modeled entities [15]. With their functionality, LQN models are ideal for modeling multi-tier systems [7]. Models could be represented as plain LQN or as XML-based LQN (LQNX), where LQN Solver (LQNS) tool [15] is used for analytically solving these models.

B. Layered Queueing eXperimenter (LQX)

LQX (Layered Queueing eXperimenter) system [15], [17] is designed and implemented to allow solving a LQN model with varied values for one or multiple parameters. For the LQN model structure which is represented in (or converted to) XML, the LQX program includes code which assigns parameter values and invokes the solver intermittently for each changed parameter value. LQX is an useful tool, considering that it employs solving the models with varied parameters quickly.

C. jLQNInterface (Java Tool)

Although LQX is a useful tool, it lacks facilities to manipulate the structure of LQN models [17]. As Mroz and Franks [17] point out, there are advantages of having a well-defined static model structure, which mainly eases “accelerated convergence” [17], thereby decreasing the time
to solve models iteratively. However, for our purposes we require manipulating LQN model structures, therefore, we have developed jLQNInterface, our own tool using Java programming language that allows solving, analyzing, and manipulating the LQNX models through its API. If models are in plain LQN model format then they are easily converted to LQNX format. Furthermore, the tool includes powerful features of object-oriented programming due to Java, is type-safe, portable and has easy to use API. The purpose of jLQNInterface is to serve as a building block for other tools and as a bridge — through its API — to allow for easy interaction with LQN models and the solver. The controller algorithm has been implemented through the jLQNInterface tool.

D. Layered Bottlenecks

"Layered bottlenecks" [8] are software or hardware resources that are saturated — i.e. they have high utilization — and which hinder performance by constraining the system throughput, as seen occurring in systems with simultaneous resource possession. Until the bottlenecks are eased performance improvements would be unnoticeable. To alleviate the bottlenecks in a system of client-server computing nodes, the options include adding software threads, increasing processors and adding new nodes (replication) of the bottleneck [16]. Specifically for VMs, replication implies instantiating copy VMs (replicas) where the incoming requests would now be equally distributed amongst the replicas.

Hardware/processor bottlenecks are easily identified by high utilization of the devices, however, software bottlenecks require a more detailed analysis. Software bottlenecks are encountered when hardware are not saturated but the system’s performance is still restricted, e.g. task bottlenecks [8]. The challenge is: not only is the particular software bottleneck saturated but other depending resources requesting services from it are also saturated because of “pushback” [8]. Furthermore, the resources that the bottleneck depends on by requesting services are unsaturated [8]. Attributes mentioned above are key towards pinpointing these bottlenecks — a process which is detailed below.

Definitions useful for finding layered bottlenecks are [8]:

\[
\begin{align*}
U_t &= \text{utilization of resource } t \\
m_t &= \text{multiplicity of resource } t \\
sat_t &= \text{saturation of resource } t \\
sat_t &= \frac{U_t}{m_t} \\
\text{BSTrength}_t &= \frac{sat_t}{sat_{\text{shadow}(T)}} \\
\text{shadow}(T) &= \arg \max_{t \in \text{Below}(T)} sat_t \\
\text{Below}(T) &= \text{the set of resources that } T \text{ depends on (directly or indirectly)}
\end{align*}
\]

Based on the paper by Frank et al. [8] we briefly describe the layered bottleneck algorithm here. Given the performance model and saturation threshold, \( sat_{\text{thresh}} \), the algorithm returns the bottleneck. The saturation of processors, \( sat_p \), and \( \text{BSTrength} \) of all other resources, which are not processors (e.g. tasks), is calculated. The processor with the highest saturation is the bottleneck if its saturation is greater than the threshold. Otherwise, if no processor is a bottleneck, then the resource with the highest \( \text{BSTrength} \) value, having its saturation greater than the threshold, is the bottleneck.

V. Provisioning Framework

In this section we describe the workings of a provisioning framework which relies on performance modeling and our controller algorithm to allocate resources through a provisioning agent (Figure 2 and Figure 3). The first step uses as input the (LQNX) performance model provided by the AP, corresponding to the initial system state. For subsequent steps, the input is the output model of the previous step. The framework invokes the controller at regular intervals — known as “controller intervals” [10] — duration of which is decided by the AP. The interval chosen should not be too close to cause frequent provisioning leading to instability and should not be too far apart to miss QoS targets. Each invocation of the controller algorithm by the framework at a given interval is considered as a step and each step comprises of multiple loops. Within each loop of a given step \( i \), the following actions are processed (Figure 2 and Figure 3):

1) **Input**: Performance model from step \( i-1 \) serves as input to step \( i \)

2) **Read/Parse**: Model is read and parsed by the Controller, which saves model information in memory.

3) **Generate**: Controller generates a copy of the model from memory and stores it on disk. This Auto model is modified through the next set of loops in the step

4) **Invoke**: LQNS is invoked to begin model solving

5) **Solve**: LQNS solves model generating a LQNX output

6) **Parse Solution**: Result is then parsed by the Controller

7) **Modify**: Based on the model, result, response time objectives, and provisioning algorithm, the controller modifies the model in-memory such that the current model state is closer to meeting QoS objectives.

8) **Repeat**: Actions 3–7 are repeated until the controller has either met QoS objectives or finds that objectives cannot be satisfied. If goals are met then action 9 is executed

9) **Provision**: Model is sent to provisioning agent

The following presents the assumptions pertaining to the provisioning framework and the controller algorithm.

A. Assumptions

A VM runs on a processor (LQN processor) of one or more multiplicity, i.e. a multi-processor, and one or more tasks may run on a VM. A VM could be replicated, distributing the incoming workload from the previous tiers between
Controller algorithm (Figure 4) begins by first initializing the internal data structures, which requires reading and parsing the input LQN model, the saturation threshold and the response time objective form the other inputs of the algorithm. The output is the model that meets the QoS. 

Algorithm: Controller SatisfyQoS

Input: InputModel, satthresh, rObjective
Output: OutputModel that satisfies QoS

Definitions:

- addedVMs: Current counter of VMs added
- rMin: approximate minimum response time possible

1: INITIALIZE()
2: rMin ← FINDMAXRESOURCEPERF()
3: if rMin > rObjective then
   4: return Unable to meet objectives
5: end if
6: INITIALIZE()
7: while true do
8:   rTime ← SOLVEMODEL()
9:   if rTime ≤ rObjective then
10:      return Objective satisfied
11:   end if
12:   bSet ← FINDBOTTLENECKS()
13:   if bSet == {} then
14:      return Unable to meet objectives
15:   end if
16:   for all B in bSet do
17:      if B ∈ Tasks then
18:         if B.processor ∈ bSet then
19:            continue
20:         else
21:            if (B.VMreplica < maxVMReplicas && addedVMs < spareVMs) then
22:               Add 1 VM replica of B.processor
23:               addedVMs + +
24:            else
25:               immutable ← immutable ∪ B
26:            end if
27:         end if
28:      else if B ∈ Processors then
29:         if B.multi < maxProcPerVM then
30:            Increase B multiplicity by 1
31:         else
32:            immutable ← immutable ∪ B
33:         end if
34:      end if
35:   end for
36: end while

Figure 4. Controller SatisfyQoS algorithm

After initialization, an approximation of maximum performance possible is found. This is accomplished by invoking the FindMaxResourcePerf algorithm. Franks et al. [8] outline a simple process of finding maximum resource performance; a similar approach is followed here. For all processors — except ones which represent client task processors and ones with infinite queueing stations (InfProc) — the multiplicity is set to the maximum of maxProcPerVM and maxVMReplicas. This corresponds to the maximum
processor multiplicity possible. Similarly, for all tasks — except the tasks which represent clients (RefTasks) and the tasks that have infinite queuing stations (InfTasks) — the multiplicity is set to the product of maxVMReplicas and the current multiplicity of the task (thread count). After these changes, the model is stored from the memory of the controller to disk and solved. The response time found is returned to the SatisfyQoS algorithm.

Once the minimum response time is received from the FindMaxResourcePerf algorithm, the SatisfyQoS verifies if the value is below the response time objective. If not, then the algorithm outputs that the model is unable to meet objectives and halts, otherwise the algorithm proceeds towards meeting the goal.

Next, the main loop of the algorithm begins; however before this the internal data structures have to be reset back to the configuration of the actual model, which is achieved by again following the initialization process. Within the loop, the model is solved by invoking LQNS and the results are parsed to find the response time. If the response time is less than the objective, then the the algorithm ends with a success. Otherwise, the bottlenecks are found by invoking the FindBottlenecks algorithm.

The FindBottlenecks algorithm (Figure 5) is a modified version of the original “layered bottlenecks” [8] algorithm (section IV-D). The main difference is use of immutable set and the isLayered boolean flag by the FindBottlenecks algorithm. This algorithm receives as input the performance model, satthresh, isLayered boolean, and immutable set. If isLayered boolean flag is true then BStrength is used for finding bottleneck from tasks after checking if a processor is the bottleneck, otherwise if false, bottlenecks are found based only on saturation values of resources, i.e. algorithm finds the layered bottleneck when isLayered flag is set to true. The immutable input contains all those resources whose multiplicity and replication cannot be increased because of reaching the limits of resources available and are therefore not considered when identifying bottlenecks. Also, processors with infinite queuing stations and tasks which are clients or have infinite queuing stations are not used in finding of bottleneck resources, as the goal here is to find bottlenecks in the cloud servers, not clients.

Any bottlenecks identified by FindBottlenecks is sent back to the SatisfyQoS algorithm. If the bottlenecks returned is an empty set, then the algorithm exits with an output that it is unable to meet objectives, otherwise the algorithm continues. For all the identified bottlenecks, it is checked if they belong to tasks or processors set. If the bottleneck is a task then the first check is to verify if the processor that the task runs on is not a bottleneck as well. If this is so, then this task bottleneck is disregarded and no action is performed on it; the reason being that any resource increments would be performed on the processor by increasing multiplicity in contrast to performing a VM replication. Otherwise, for the bottleneck task, if it’s processor is not the bottleneck, then replication of the task’s VM is performed. However, if the bottleneck is a processor then the multiplicity of the processor is incremented. Before performing VM replication or before adding processors, the limits: spareVMs, maxProcPerVM, maxVMReplicas are checked, and if any limit is reached then instead of adding resources, the resource is noted as immutable. This information is later used when finding bottlenecks as detailed earlier.

Above describes one complete loop of the algorithm, and the next loop begins again with solving the performance model, finding bottlenecks and adding resources until either no bottlenecks remain — in which case the algorithm

---

**Algorithm: FindBottlenecks**

**Input:** InputModel, satthresh, isLayered, immutable

**Output:** Bottleneck set: bSet

1. for all \( p \in \text{Processors} \) where \( p \notin \text{InfProc} \) do
   1. Calculate \( sat_p \)
   1. end for
2. for all \( t \in \text{Tasks} \) where \( t \notin \text{RefTask}, \text{InfTask} \) do
   1. Find \( \text{Below} \) and Calculate \( sat_t \)
   1. Calculate \( BStrength_t \)
   1. end for
3. if \( \text{isLayered} == \text{true} \) then
   1. \( B \leftarrow \arg \max_{p \in \text{Processors}} sat_p \)
   1. where \( sat_p \geq sat\_thresh \) && \( B \notin \text{immutable} \)
   1. if \( B \neq \text{NULL} \) then
   1. \( bSet \leftarrow bSet \cup B \)
   1. end if
   1. else
   1. \( B \leftarrow \arg \max_{t \in \text{Tasks}} BStrength_t \)
   1. where \( sat_t \geq sat\_thresh \) && \( B \notin \text{immutable} \)
   1. if \( B \neq \text{NULL} \) then
   1. \( bSet \leftarrow bSet \cup B \)
   1. end if
   1. end if
4. for all \( p \in \text{Processors} \) do
   1. if \( sat_p \geq sat\_thresh \) then
   1. \( bSet \leftarrow bSet \cup p \)
   1. end if
   1. end for
5. for all \( t \in \text{Tasks} \) do
   1. if \( sat_t \geq sat\_thresh \) then
   1. \( bSet \leftarrow bSet \cup t \)
   1. end if
   1. end for
6. end if
7. return \( bSet \)

---

Figure 5. FindBottlenecks algorithm
ends with output of unable to meet objectives — or the desired response time objective is met — in which case the algorithm ends with output of meeting objectives. In the latter case, the final performance model would be sent to the provisioning agent to allocate resources.

VI. CASE STUDY

A case study using the SatisfyQoS algorithm was conducted using a simple LQNX input model with aim to demonstrate the applicability of the proposed algorithm. One complete step of the algorithm, involving several loops, was run such that desired response time objective could be achieved. The following two cases were tried:

1) isLayered true: layered bottlenecks
2) isLayered false: bottlenecks based on saturation only

isLayered boolean has been discussed in section V-B. The input model, the output models and the result from the two cases have been presented below. LQNS version 5.4 was used for the evaluations.

Figure 6 shows the input model representing 100 Browser clients and four uni-processor VMs on the cloud, each running a single task. Each Browser request is sent first to TierA. For a received request, multiple requests are sent further down to another task and this value is shown within square brackets and the maximum thread count is shown within curly braces. The parameters were set as follows: \textit{sat\textsubscript{threshold}} = 0.8, \textit{spareVMs} = 20, \textit{maxVMReplicas} = 10, \textit{maxProcsPerVM} = 6. The input model response time was 1806.720 ms and desired response time was 350 ms.

For case 1, results from running the algorithm are shown in Table I. The algorithm executed 15 loops and performed 19 changes in total to meet the response time objective. The response time of the Browser task was 302.942 ms. The output model included 14 VMs and 27 processors.

For case 2, results from running the algorithm are shown in Table II. The algorithm executed 5 loops and performed 19 changes in total to meet the response time objective. The response time of the Browser task was 299.537 ms. The output model included 14 VMs and 27 processors.
to increase of resources one at a time rather than multiple resource increments as within each loop of case 2. But, case 1 caused fewer changes and resulted in use of far less VMs and processors than case 2. A possible reason is due to the search for bottlenecks after each change and selective addition in case 1, where preference is given to only one resource with maximum saturation or **BStrength** value. Based on the results, case 1 met the objectives with fewer resources.

**VII. Conclusions**

In this paper, a dynamic provisioning algorithm for adding resources is proposed that can be applied to cloud computing domain. The algorithm finds bottlenecks based on the notion of “layered bottlenecks” [8]. Depending on bottleneck type, either processors or VMs are added to help ease the bottleneck, thereby improving performance. Before incrementing, the limits such as spare VMs, processors-per-VM, and replication-per-VM are checked and kept under bounds. This process of finding bottlenecks and adding resources is continued until the desired performance objectives are satisfied.

The provisioning algorithm has been implemented by using jLQNInterface, a multi-purpose tool developed by the authors for solving, analyzing, and manipulating the LQN models. Furthermore, the jLQNInterface API serves as a building block and bridge for other utilities that intend on interfacing with LQN models and the solver.

A case study demonstrates the applicability of our proposed algorithm and use of layered bottlenecks in resource provisioning through a run of the implemented algorithm. Two cases were tried, where case 1 used layered bottlenecks and case 2 used saturation values to determine the bottlenecks. It is found that although case 1 required more loops through the algorithm, the resulting model used fewer resources while meeting the desired objectives.

Future work would incorporate deallocation of VMs and processors when utilizations are below a given threshold. Alongside, instead of having single replication limit for all VMs, different limits could be added specific to each application-tier. Currently response time of one client task is considered, in future, the aim is to support meeting of response times of multiple client tasks. Optimizing the cost of the resources in the algorithm would also be a considered in future works.

**Acknowledgment**

The authors would like to thank NSERC for their support.

**References**


