A Receding Horizon Approach for the Runtime Management of IaaS Cloud Systems

www.modaclouds.eu

Danilo Ardagna, Michele Ciavotta
{danilo.ardagna, michele.ciavotta}@polimi.it
Politecnico di Milano

Riccardo Lancellotti
riccardo.lancellotti@unimore.it
Università di Modena e Reggio Emilia
* Introduction

* Problem
  * Problem statement and design assumption
  * Receding Horizon algorithm

* Experimental Analysis

* Conclusions
The advent of Cloud Computing changed dramatically the ICT industry

* Google, Amazon, Microsoft, Salesforce, Oracle, SAP, SoftLayer, Rackspace etc...
* Cost-effective solutions
* Computational power
* Reliability
* Auto-scaling

New business paradigms appeared on the market

* IaaS, PaaS, SaaS
* But also DaaS, BDaaS, HDaaS, etc...
The growing popularity of Cloud Computing opens new challenges

* Vendor lock-in
* Design for Quality of Service (QoS) guarantees
* Managing the lifecycle of a Cloud application
* Managing Elasticity
  * Resource Provisioning
  * Self-adaptation
Introduction: resource provisioning

Resource Provisioning: mechanism for leasing and releasing virtual cloud resources to guarantee adequate QoS

... it requires management solutions that support

* Performance prediction,
* Monitoring of Service Level Agreements (SLA),
* Adaptive re-configuration actions.

Tools currently supplied by IaaS providers, are often too basic and inadequate for

* Highly variable workload,
* Applications with a dynamic behavior characterized by uncertainty.
Introduction: our approach

Proposal: a fast and effective Capacity Allocation technique

* based on the Receding Horizon control strategy
* integrated within MODAClouds runtime platform
* that minimizes the execution costs of a Cloud application,
* guaranteeing QoS constraints expressed in terms of average response time
Agenda

* Introduction

* Problem
  * Problem statement and design assumption
  * Receding Horizon algorithm

* Experimental Analysis

* Conclusions
Problem: design assumptions

Perspective of a **Software-as-a-Service (SaaS)** provider hosting his/her applications on an **Infrastructure-as-a-Service (IaaS)** provider

**Applications** are single **tier** hosted in virtual machines (VMs) that are dynamically instantiated by the IaaS provider

Each VM hosts a single **WS application**

Multiple **homogeneous** VMs implementing the same WS application can run in parallel
Problem: design assumptions

Each **WS class** hosted in a VM is modeled as an **M/G/1 queue** in tandem with a delay center

**SLA** based on the average response time: every WS class has to provide a response time \( R_k \) lower than a threshold \( \overline{R}_k \)
IaaS providers **charge** software providers on an **hourly basis**

- *reserved VMs* (ρ time-unit cost)
- *on demand VMs* (δ time-unit cost; ρ < δ)

**Problem:** design assumptions

Time management:
- **Time slots:** $T_{slot}$ (5, 10 min)
- **Time window:** $T_w$ (1-5 $T_{slot}$)
- **Charging interval:** $T_c$ (60 min)
### Problem: formulation

#### System parameters

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$K$</td>
<td>Set of WS applications</td>
</tr>
<tr>
<td>$\delta$</td>
<td>Time unit cost (measured in dollars) for on-demand VMs</td>
</tr>
<tr>
<td>$\rho$</td>
<td>Time unit cost (measured in dollars) for reserved VMs</td>
</tr>
<tr>
<td>$T_w$</td>
<td>Set of time slots within the sliding time window</td>
</tr>
<tr>
<td>$T_c$</td>
<td>Set of time slots within a charging interval</td>
</tr>
<tr>
<td>$T_{slot}$</td>
<td>Short-term CA time slot, measured in minutes</td>
</tr>
<tr>
<td>$n_c$</td>
<td>Number of time slots within the charging interval $T_c$</td>
</tr>
<tr>
<td>$n_w$</td>
<td>Number of time slots within the time window $T_w$</td>
</tr>
<tr>
<td>$\bar{r}_k^t$</td>
<td>Number of reserved VMs freely available at time slot $t$ in the interval under analysis, for request class $k$</td>
</tr>
<tr>
<td>$\bar{d}_k^t$</td>
<td>Number of on-demand VMs available for free at time slot $t$ in the interval under analysis, for request class $k$</td>
</tr>
<tr>
<td>$\lambda_k^t$</td>
<td>Real local arrival rate (measured in requests/sec) for request class $k$, at time slot $t$</td>
</tr>
<tr>
<td>$\bar{\lambda}_k^t$</td>
<td>Local arrival rate prediction (measured in requests/sec) for request class $k$, at time slot $t$</td>
</tr>
<tr>
<td>$R_k$</td>
<td>Average response time threshold for request class $k$</td>
</tr>
<tr>
<td>$W$</td>
<td>Maximum number of reserved instances available</td>
</tr>
</tbody>
</table>

#### Decision Variables

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_k^t$</td>
<td>Number of on-demand VMs to be allocated for request class $k$ at time slot $t$</td>
</tr>
<tr>
<td>$r_k^t$</td>
<td>Number of reserved VMs to be allocated for request class $k$ at time slot $t$</td>
</tr>
</tbody>
</table>
The CA problem can be formulated as:

\[(P) \quad \min_{r_k^t, d_k^t} \sum_{k \in \mathcal{K}} \left( \rho \sum_{t=1}^{n_w} r_k^t + \delta \sum_{t=1}^{n_w} d_k^t \right)\]

Subject to the conditions:

\[R_k(r_k^1, \bar{r}_k^1, \ldots, r_k^t, \bar{r}_k^t, d_k^1, \bar{d}_k^1, \ldots, d_k^t, \bar{d}_k^t, \hat{\Lambda}_k^1, \ldots, \hat{\Lambda}_k^t) \leq \bar{R}_k\]
\[\forall k \in \mathcal{K}, \forall t \in \mathcal{T}_w\]

\[\sum_{k \in \mathcal{K}} (r_k^t + \bar{r}_k^t) \leq W, \forall t \in \mathcal{T}_w\]

\[r_k^t \geq 0, \quad r_k^t \in \mathbb{N} \quad \forall k \in \mathcal{K}, \forall t \in \mathcal{T}_w\]

\[d_k^t \geq 0, \quad d_k^t \in \mathbb{N} \quad \forall k \in \mathcal{K}, \forall t \in \mathcal{T}_w\]
In a nutshell, the Capacity Allocation problem is solved for every time slot in $T_w$ but only the actions concerning the first forthcoming time slot are enacted.
Receding Horizon Algorithm

First slot configuration \((r_k^1, d_k^1)\)

Optimizer

Optimization Model \(T_w\)

Solve

Optimal solution

Receding Horizon controller

Clock

Update Model Parameters

Predicted workload \((\hat{\Lambda}_k^1, \ldots, \hat{\Lambda}_k^{p_w})\)

IaaS Interface

Monitoring Platform

Cloud Application

 MODA Clouds
# Receding Horizon Algorithm

## Algorithm 1 Receding Horizon Algorithm

1. **procedure** SOLUTION_ALGORITHM
2.   **for all** \( k \in \mathcal{K} \) **do**
3.     **for** \( w \leftarrow 1, n_w \) **do**
4.       \( \hat{\Lambda}_w \leftarrow \text{GetPrediction} \ (w, k) \)
5.       \( \hat{r}_k^w \leftarrow N_{res,k}^{t+w} \)
6.       \( \hat{d}_k^w \leftarrow N_{ond,k}^{t+w} \)
7.     **end for**
8.   **end for**
9. **Solve** \((P, \bar{r}, \bar{d}, \hat{\Lambda})\)
10. **for all** \( k \in \mathcal{K} \) **do**
11. **Scale** \((k, r_k^{1}, d_k^{1})\)
12.   **for** \( j \leftarrow 1, n_c \) **do**
13.       \( N_{res,k}^{t+j} \leftarrow N_{res,k}^{t+j} + r_k^{1} \)
14.       \( N_{ond,k}^{t+j} \leftarrow N_{ond,k}^{t+j} + d_k^{1} \)
15.   **end for**
16. **end for**
17. **end procedure**
Agenda

- Introduction

- Problem
  - Problem statement and design assumption
  - Receding Horizon algorithm

- Experimental Analysis

- Conclusions
Experimental Analysis

Scalability:

* Large set of randomly generated instances
* Daily distribution of requests from real log traces

Comparison with state of the art approaches:

* Heuristic
* Oracle with perfect knowledge of the future

Time scale analysis:

* SLA violations
Workload prediction

- Incoming workload has been obtained for traces of a very large dynamic web-based system
- Different workload for each WS class
- Prediction obtained by adding white noise to each sample
- Noise proportional to the arrival rate
- Inaccuracy increases with the time slot

Performance parameters

- Service rate $\mu_k \in [200, 400]$ req/sec
- Queueing delay $D_k \in [0.001, 0.05]s$
- Reserved instances $W = 10$

Instance cost

- Randomly generated considering prices currently charged by common IaaS providers
Traffic profiles:

- Normal workload with low noise
- Normal workload with high noise
- Spiky workload with low noise
- Spiky workload with high noise

The different levels of noise correspond to:
Scalability

The analysis demonstrated that our approach scales almost linearly with respect to the number of request classes. Systems up to 160 classes and 5 time slots can be solved in less than 200 sec.
Cost – Normal traffic

Costs comparison

<table>
<thead>
<tr>
<th>Solution</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>S-t Algorithm</td>
<td>2.00%</td>
<td>5.78%</td>
<td>9.31%</td>
<td>13.05%</td>
</tr>
<tr>
<td>Heu (40%, 50%)</td>
<td>95.81%</td>
<td>68.97%</td>
<td>85.51%</td>
<td>122.15%</td>
</tr>
<tr>
<td>Heu (50%, 60%)</td>
<td>95.09%</td>
<td>63.72%</td>
<td>85.59%</td>
<td>122.88%</td>
</tr>
<tr>
<td>Heu (60%, 80%)</td>
<td>52.39%</td>
<td>34.88%</td>
<td>47.54%</td>
<td>65.81%</td>
</tr>
</tbody>
</table>

10 minutes time scale
- Low noise level
Cost – Spiky traffic

Costs comparison

<table>
<thead>
<tr>
<th>Solution</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>S-t Algorithm</td>
<td>5.10%</td>
<td>7.75%</td>
<td>11.02%</td>
<td>13.73%</td>
<td>16.86%</td>
</tr>
<tr>
<td>Heu (40%, 50%)</td>
<td>175.37%</td>
<td>145.68%</td>
<td>162.70%</td>
<td>195.95%</td>
<td>210.97%</td>
</tr>
<tr>
<td>Heu (50%, 60%)</td>
<td>197.36%</td>
<td>165.89%</td>
<td>183.58%</td>
<td>216.52%</td>
<td>229.00%</td>
</tr>
<tr>
<td>Heu (60%, 80%)</td>
<td>138.10%</td>
<td>115.46%</td>
<td>126.85%</td>
<td>138.88%</td>
<td>146.38%</td>
</tr>
</tbody>
</table>

5 minutes time scale
- Low noise level
Goal:
  evaluate the impact of time scale on the proposed receding horizon algorithm.

Analyses have been supported by a discrete event simulator based on the Omnet++ framework created on purpose. Able to capture the time-varying performance degradation due to resource contention via Random Environments (REs).

Performance indicators considered:
- SLA violation (the percentage of time slots where the average response time exceeds the SLA thresholds)
- Dropped request (the percentage of requests dropped as a result of the finite queue length)

Time scale analysis

![Time scale analysis graph](image-url)
## Time scale analysis

<table>
<thead>
<tr>
<th>$\tau_w$</th>
<th>SLA Violations [%]</th>
<th>Dropped Requests [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5 min</td>
<td>10 min</td>
</tr>
<tr>
<td>1</td>
<td>0.49</td>
<td>1.74</td>
</tr>
<tr>
<td>2</td>
<td>1.08</td>
<td>0.56</td>
</tr>
<tr>
<td>3</td>
<td>0.90</td>
<td>1.81</td>
</tr>
<tr>
<td>4</td>
<td>1.15</td>
<td>1.88</td>
</tr>
</tbody>
</table>

The values are related to a 24 hours analysis with low noise and averaged over 10 executions.

A control time granularity of 5 minutes tends to provide better performance if compared to granularity of 10 minutes both in terms of SLA violations and in terms of dropped requests.
* Introduction

* Problem
  * Problem statement and design assumption
  * Receding Horizon algorithm

* Experimental Analysis

* Conclusions
We proposed optimization approach to achieve fast, scalable and effective capacity allocation based on a fine grained time scale.

Our technique is able to minimize costs in a more efficient way than the current state of the art.

The QoS defined into the SLA is almost always respected (less than 2% and 7 min).

Future works:

* development of an adaptive approach able to switch between different time scales according to the workload conditions

* Test on a real prototype environment
Thank You!

Questions?
MODAClouds: MOdel-Driven Approach for design and execution of applications on multiple Clouds

Focus on needs of Cloud-based Application Developers and Operators

Objective: to provide methods, a decision support system, an IDE and a runtime environment to support

- High-level design
- Semi-automatic code generation
- Automatic (re)deployment of applications on multi-Clouds
- Self-adaptive mechanism to guarantee Service Level Agreements (SLAs) with end users