International Journal of Innovative Research in Advanced Engineering (IJIRAE)
Issue 2, Volume 2 (February 2015)
ISSN: 2349-2163
www.ijirae.com

# Cost Effective Automated Scaling of Web Applications for Multi Cloud Services

**SANTHOSH.A[1], D.VINOTHA[2], BOOPATHY.P[3]**
[1,2,3] *Computer Science and Engineering*
*PRIST University*
*India*

***Abstract - Resource allocation in cloud computing is traditionally centralized for limits the multi cloud server services. The most interesting feature that Cloud Computing brings, is the on-demand resource provisioning model. Many web applications can benefit from an automatic scaling property where their resource usage can be scaled up and down automatically by the multi cloud service. In this paper, we propose a novel solution that automatic scaling mechanism designed towards web applications in the multi cloud service environment for scalable resource management scheme. Achieving the above mentioned is not trivial and done by Cost Effective Automated Scaling (CEAS), which is belongs to the family of color set algorithm. The paper demonstrates how the above solutions can be integrated into an overall design for a web application of Multi cloud service that exhibits properties of self auto scaling.***

***Keywords: Eigen Values, Neural Networks, PCA, Artificial Intelligence,***

## I.INTRODUCTION

SELF AUTO- SCALABILITY capabilities are essential for large scale cloud servers and the services they provide, in order to keep configuration complexity low and to achieve performance objective in a changing environment. Special attention in this context must be devoted to the design of resource management mechanisms, as current solutions do not sufficiently scale. Engineering a scalable resource management system for cluster-based services includes addressing two key problems. The first is Application Placement, which refers to developing an approach for allocating the system resources to a set of applications that the cluster offers, by deciding which application should run on which node. The allocation must be efficient (in this work, the utilization of the computational resources of the cluster must be maximized) and must take into account the application requirements (in this work, CPU and memory requirements), the quality-of-service objectives of the clients (in this work, the maximum response time) and the policies for service differentiation under overload (in this work, the relative importance of the applications). The second problem relates to Request Routing, which directs service requests to available resources inside the cluster. The routing scheme must be efficient and ensure a balanced load in the system. In this work, we specifically address the problem of developing a routing scheme whose
over head is independent of the system size.

The architecture of our system is shown in Fig. 1. We encapsulate each application instance inside a virtual machine (VM). The use of VMs is necessary to provide isolation among untrusted users. Both Amazon EC2 and Microsoft Azure use VMs in their cloud computing offering. Each server in the system runs the Xen hypervisor which supports a privileged domain 0 and one or more domain U .

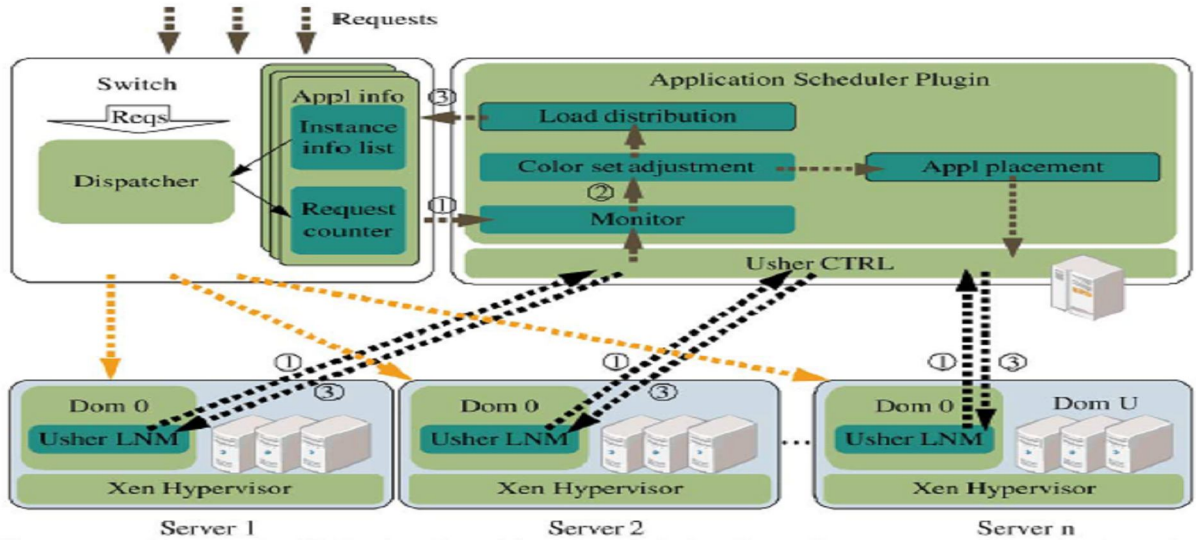



Fig. 1: Overview of the architecture

Resource containers are first class operating system abstraction for resource principles; it is independent of process and threads. Using resource containers, a web server can associate, for instance, a client network connection, a server thread, and a CGI process with a single resource principle that represent client request being served. This principle competed with other principles representing other client requests for server resources. Resource container allow accurate accounting and scheduling of resource on behalf of a single client requests or a class of client requests, and enable performance isolation and differentiated quality of service when combined with an appropriated resource scheduler [2].

A major challenge for Web farms is to balance the load on the servers effectively, so as to minimize the average response time on the system. Overutilization of servers can cause excessive delays of customer requests. On the other hand, underutilization of servers is wasteful. Minimizing average response time will have the dual effect of maximizing the throughput the clustered Web farm can achieve. This means that the load-balance mechanism the front-end dispatcher can be made more permissive [25].

We give a use case for our work and introduce the principles that guide the overall design. In Section II we provide details of system background. Section III shows our CEAS for automatic scaling and describes how to solve auto scaling of web application in multi cloud environment. In Section IV we describe the implementation of the design. In Section V we evaluate the design through simulation. Finally, in Section VI we conclude the paper

## II. SYSTEM BACKGROUND

Achieving scalability in the number of applications in an efficient way mandates that each node in the cluster can run multiple applications.

Cloud scaling proposals all are based on the condition actions approach, they vary substantially in several aspects: which metrics are monitored; expressiveness of the conditions defining mechanism; and which actions can be taken. Many of them focus on horizontal scaling, [3] i.e., deploying or releasing VMs, while vertical scaling(like for example increasing physical resources of an overloaded server) is not considered, possibly due to the impossibility of changing the available CPU, memory, etc., on-the-fly in general purpose OSs. Here we analyze the different auto-scaling solutions used by several cloud proposals, commercial ones such as Amazon EC2 and open source solutions.

The Class Constrained Multiple Knapsack problem(CCMK) aims to maximize the total number of packed items under the restriction that each knapsack has a limited capacity and a bound on the number of different types of items it can hold [4], [5].Unlike CCBP, it does not attempt to minimize the number of knapsacks used. Hence, unlike our algorithm, it does not support green computing when the system load is low.

Resource provisioning for Web server farms has been investigated in [23]. Some allocate resources in the granularity of whole servers which can lead to inefficient resource usage. Some do not consider the practical limit on the number of applications a server can run simultaneously. Bhuvan et al. supports shared hosting, but manage each application instance independently. They do not provide the auto scaling property. Mohit et al. group applications into service classes which are then mapped onto server clusters [24]. However, they do not attempt to minimize the placement changes when application demands vary and are mostly for offline use. Zhang et al. organize a set of shared clusters into a network and study resource allocation across shared clusters, which is not the focus of this paper.

## III. CEAS FOR AUTOMATIC SCALING OF RESOURCES

Load balancing is not a unique feature to cloud platforms; it should not be regarded as independent from auto-scaling. In fact, the two need to work together in order to get most efficient platform usage and save expenses.

The end goal of load balancing from the cloud's client point of view is to have a more efficient use of the virtual resources that has running and reduce cost. That load balancing should be used in conjunction with auto-scaling in order to reduce cost. As s result we have the following usage scenarios:

1. **High platform load** when the cloud client's overall platform load is high, as defined by the cloud client. The platform needs to scale up by adding more virtual resources. The load balancing element automatically distributes load to the new resources once they are registered as elements of the platform and therefore reduces platform load.
2. **Low platform load** when the cloud client's overall platform load is low, as defined by the cloud client. In this situation, the platform needs to scale down by terminating virtual resources. This is not as trivial as the previous scenario, because the load balancing element typically assigns tasks to all resources and therefore prevents resources from reaching a state where their load is zero and can be terminated. In this situation, the load balancing element needs to stop distributing load to the part of the platform that will be released and, even more, currently- running tasks of this part of the platform needs to be migrated to ensure that a part of the platform will have zero load therefore can be released.

### A) Details of Our Algorithm

The resource requirement and management of web application as individual elements in the cloud server and use resource bin packing to solve our problem [1]. Unfortunately, the resource requirement of web applications has to be satisfied as a whole: a major portion of the resource is consumed anyway even when the application receives little load. This is especially true for Java applications. None of the existing bin packing problems can be applied in our environment. Our algorithm belongs to the family of size set algorithms, but with significant modification to adapt to our problem. We label each class of items with a color and organize them into color sets as they arrive in the input sequence. The number of distinct colors in a color set is at most c (i.e., the maximum number of distinct classes in a bin). This ensures that items in a color set can always be packed into the same bin without violating the class constraint. The packing is still subject to the capacity constraint of the bin. All color sets contain exactly c colors except the last one which may contain fewer colors.

### B) The color set Algorithm

Consider a simple algorithm $A_{cs}$, which partitions the M σ, colors is ω into [M σ/c] color -sets and packs, the items of each color-sets greedily. Each color- sets consist of C colors (excluding the last color-set that may contain fewer colors). The partition into color-sets in determined online by the input sequence. That is, the first set, C1, consists of the first C colors in σ, the second set, C2, of the next C colors in σ and so on. At any time, there is one active bin for each color set. When an item ă color i€Cj arrives, it is placed in the bin of $C_j$, if the arrives bin contain υ items, we open a new bin for ă and this is the new active bin of $C_j$. Since $|C_j| \leq c$, the resulting placement is feasible.

Theorem for $A_{cs}$, the color sets algorithm, $r_{cS}<2$.

Proof: Assume that when Acs terminates there are l active bins, containing $X_1$……. $X_l$ items, since we open a new bin for some color –set only when the current active bin of that color set is full, we have

$$N_{CS}(\sigma) = \frac{n - (x_1 + x_2 +, \ldots, x_\ell)}{v} + \ell \leq \frac{n}{v} + \ell\left(1 - \frac{1}{v}\right).$$

### C) Application Load Increase

The load increase of an application is modeled as the arrival of items with the corresponding olor. A naive algorithm is to always pack the item into the unfilled bin if there is one. If the unfilled bin does not contain that color already, then a new color is added into the bin. This corresponds to the start of a new application instance which is an expensive operation. Instead, our algorithm attempts to make room for the new item in a currently full bin by shifting some of its items into the unfilled bin. Let be the color of the new item and be any of the existing colors in the unfilled bin. We search for a bin which contains items of both colors. Let be such a bin. Then we move an item of color C2 from bin to the unfilled bin. This makes room for an item in bin where we pack the new item.If we cannot find a bin which contains both colors, we see if we can shift the items using a third color C3 as the intermediate. More specifically, we search for two bins:

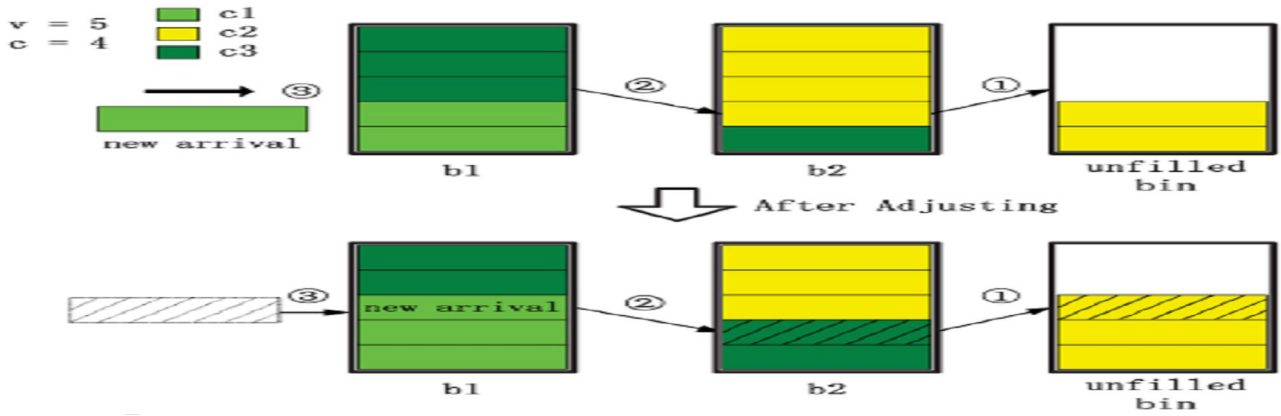
bin b1 contains c1 colors and c3
bin b2 contains c2 colors and c3
If we can find such two bins, we proceed as follows:
move an item of color c2 from bin b2 to the unfilled bin
move an item of color c3 from bin b1 to bin b2
pack the item in bin b1

This process is illustrated in Fig. 2. (Recall that v is the capacity of the bin and c is the class constraint.) More generally, we can have a chain of colors c1, ,ck such that

- c1 is the color of the new item
- ck is an existing color in the unfilled bin
- every two adjacent colors in the chain share a bin2

The length of the chain is bounded by the number of colors in the color set (i.e., the class constraint). As long as such a chain exists, we can accommodate then few item by shifting the existing items along the chain. Note that the item movements here are hypothetical and used only to calculate the new load distribution. No physical movement of any application occurs. Also note that the chain length is bounded by a constant and does not increase with the numbers of applications or servers in the system. If we cannot find such a chain, the new color has to be added into the unfilled bin which requires starting anew application instance. If the color set has no unfilled bin, then a new bin is allocated. If all bins are used up, then the load increase cannot be satisfied.

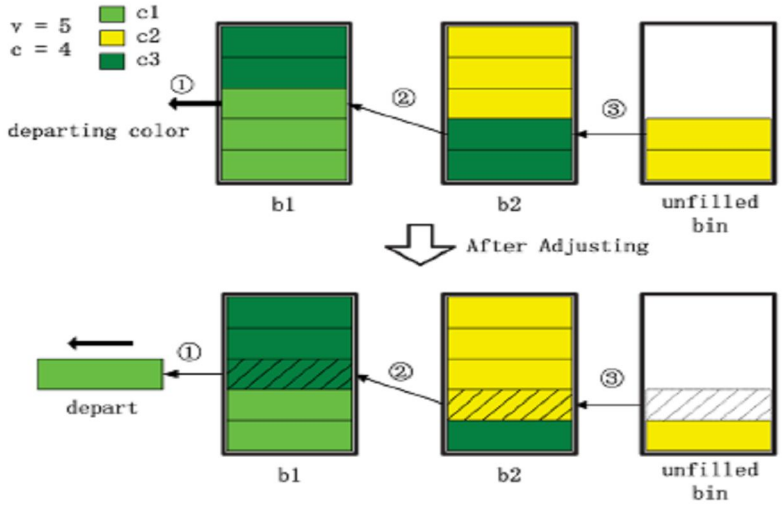



1→ Move an item of color c2 from bin b2 to the unfilled bin
2→ Move an item of color c3 from bin b1 to bin b2
3→ Pack the new arrival in bin b1

Fig .2. Arrival of new item

**D) Application Load Decrease**

The load decrease of an application is modeled as the departure of previously packed items. Note that the departure event here is associated with a specific color, not with a specific item. The algorithm has the freedom to choose which item of that color to remove. The challenge here is to maintain the property that each color set has at most one unfilled bin. Our departure algorithm works as follows. If the color set does not have an unfilled bin, we can remove any item of that color and the resulting bin becomes the unfilled bin. Otherwise, if the unfilled bin contains the departing color, a corresponding item there can be removed directly. In all other cases, we need to remove an item from a currently full bin and then fill the hole with an item moved in from somewhere else. Let c1 be the departing color and c2 be any of the colors in the unfilled bin. We need to find a bin which contains items of both colors. Let b be such a bin. We remove the departing item from b bin and then move in an item of color c2 from the unfilled bin. More generally, we can find a chain of colors and fill the hole of the departing item by shifting the existing items along the chain. The procedure is similar to the previous case for application load increase. Fig. 3 illustrates this process for a chain with three colors. If we cannot find such a chain, we start a new application instance to fill the hole: remove an item of the departing color from any bin which contains that color. select a color C2 in the unfilled bin and add that color into the departing bin. move an item of color C2 from the unfilled bin to the departing bin. If the unfilled bin becomes empty, we can remove it from the color set and shut down the corresponding server since all application instances there receive no load




1→ Remove an item corresponding to the departing color
2→ Move an item of color c3 from b2 to bin b1
3→ Move an item of color c2 from the unfilled bin to bin b2

Fig 3: departure of existing system

## IV. EXPERIMENTS

We evaluate the effectiveness of our system in experiments. The Web applications used in the experiments are Apache servers serving CPU intensive PHP scripts. Each application instance is encapsulated in a separate VM.

### A) Auto Scaling

We evaluate the auto scaling capability of our algorithm with nine applications and 30 Dell Power Edge servers Fig. 7 (left) shows the request rate of the flash crowd application and the number of active servers (i.e., APMs) used by all applications over the course of the experiment. Initially, the load in the system is low and only a small number of servers are used. When the flash crowd happens, our algorithm detects the skyrocketing request rate quickly and scales up the server resources decisively.

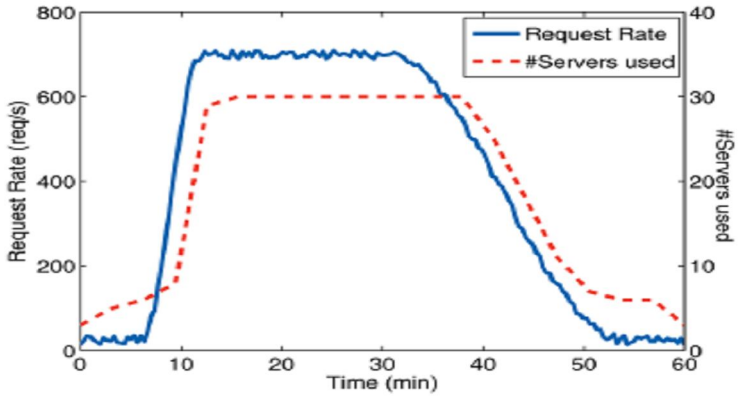



Fig.4. Request rate and server used

## V. SIMULATIONS

The previous section has demonstrated the effectiveness of our system in real experiments. This section evaluates the performance of our application scheduling algorithm in large scale simulation. Our simulator uses the same code base for the scheduling algorithm as the real implementation in the previous section. This ensures the fidelity of our simulation results.

### Scalability

We evaluate the scalability of the algorithm by increasing both the number of servers and the number of applications from 1000 to 10,000. Fig. 5 shows how the decision time increases with the system size. As we can see from the figure, our algorithm is extremely fast, even when implemented in Python: the decision time is less than 4 seconds when the system size reaches 10,000. The middle figure shows that the demand satisfaction ratio is independent of the system size (depends on D as shown previously). The right figure shows that the number of placement changes increases linearly with the system size. Again the class constraint has a bigger impact when the demand is higher. When averaged over the number of servers, each server experiences roughly a constant number of placement changes for any given demand ratio

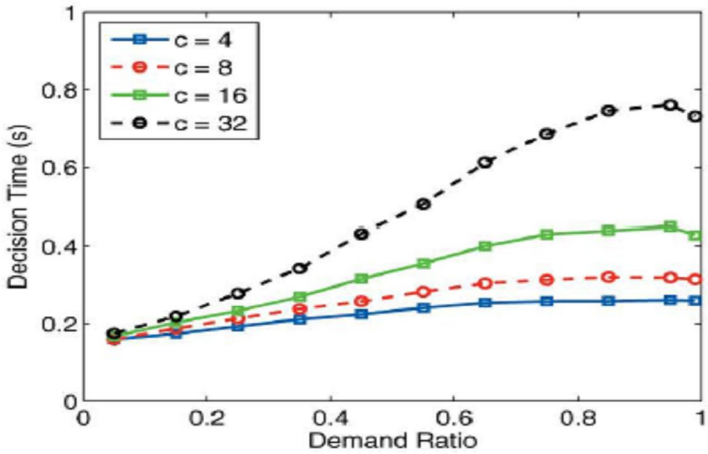



Fig 5: performance of the algorithm

The average decision time in each run of the algorithm is shown in Fig. 5. Note that the decision time is based on the real execution of the actual code used in the deployment. The figure indicates that the decision time increases with the demand ratio and with the class constraint.

## VI.CONCLUSION

The resource allocation in multi cloud computing infrastructures Compared with a previous scaling methodologies, cost effective auto scaling balance the load across web machines. In this paper we develop the cost effective auto scaling based color set algorithm to decide the application placement and load distribution. Our system achieves high satisfaction ratio of application demand even when the load is very high. It saves energy by reducing the number of running instances when the load is low.

REFERENCES

[1] Zhen Xiao, Senior Member, IEEE, Qi Chen, and Haipeng Luo "*Automatic Scaling of Internet Applications for Cloud Computing Services*" IEEE TRANSACTIONS ON COMPUTERS, VOL. 63, NO. 5, MAY 2014

[2] Mohit Aron Peter Druschel Cluster Reserves: A Mechanism for Resource Management in Clusterbased Network Servers "*international parallel processing symposium* 1999.

[3] https://hal.inria.fr/hal-00668713 "*Auto-Scaling, Load Balancing and Monitoring in Commercial and Open-Source Clouds"*

[4] Chongjie Zhang, Victor Lesser, Prashant Shenoy *"A Multi-Agent Learning Approach to Online Distributed Resource Allocation"* In ICML'03, pages 928–936, 2003.

[5] Chunqiang Tang, Malgorzata Steinder, Michael Spreitzer, and Giovanni Pacifici *"A Scalable Application Placement Controller for Enterprise Data Centers "* WWW 2007 / Track: Performance and Scalability.

[6] Constantin Adam, *Student Member, IEEE,* and Rolf Stadler, *Member, IEEE "Service Middleware for Self-Managing Large-Scale Systems"* IEEE TRANSACTIONS ON NETWORK AND SERVICE MANAGEMENT, VOL. 4, NO. 3, DECEMBER 2007.

[7] Riccardo Lancellotti, Mauro Andreolini, Claudia Canali, Michele Colajanni, *"Dynamic request management algorithms for Web-based services in cloud computing"* University of Modena and Reggio Emilia, 2011, international conference .