Automated Capacity Management and Selection of Infrastructure-as-a-Service Providers

Alexander Stage, Thomas Setzer, and Martin Bichler
Technische Universität München (TUM)
Department of Informatics (I18)
Boltzmannstr. 3, 85748 Garching, Germany
{stage, setzer, bichler}@in.tum.de

***Abstract*—Infrastructure-as-a-service (IaaS) providers are gaining popularity in the application hosting market. Virtual machines of different sizes (capacities for server resources such as CPU, main memory, etc.) are offered on-demand on an hourly, daily, weekly or monthly basis. Which offering minimizes the hosting costs depends on the sizes and respective prices, but also on the demand patterns of an application and is a non-trivial selection problem for the customer. The fact that IaaS providers have different minimum subscription times, makes the problem particularly hard. In this paper, we describe an optimization formulation that determines an optimal schedule of virtual machines, which is needed to satisfy the demand of a particular application. The algorithm can be used to select the cost minimal offering of different hosting providers, but also to control the allocation and de-allocation of virtual machines with an IaaS provider over time.**

I. Introduction

Efficient capacity management for business applications is a challenging task since most applications' resource demand is characterized by high peak-to-mean-ratios, recurring and shifting seasonal patterns as well as bursts leading to heavy-tail demand distributions [1], [2]. Therefore, to ensure efficient delivery of acceptable application quality over time without over-provisioning, continuously carried out capacity management activities are required [3], [4].

A key property of most business applications is their ability to scale-out almost arbitrarily by relying on load sharing and balancing techniques, which requires hosting infrastructures that allow for on-demand allocation of additional cluster nodes.

Cloud infrastructure providers like Amazon EC2 [1], GoGrid [2] or SliceHost [3] promise to deliver the required dynamic resource provisioning facilities for scalable application hosting. Several vendors are currently emerging, that allow the deployment of customer managed virtual machines (VM). The different offerings feature varying pricing models for fixed sized VMs (capacities for CPU, main memory, etc.), SLAs and additional services such as load balancing. Available VM sizes range from small instances (single virtual CPU ranging from 1.0 to 3.0 Ghz in combination with main memory entitlements of 256 MB to 1.7 GB) up to large instances with multiple virtual CPUs (with aggregated capacities between 12 to 24 Ghz) and main memory entitlements of 8 to 32 GB. Prices of a provider increase almost linearly with VM sizes for most offerings. A second commonality between all providers is the existence of minimum subscription times, typically an hour, a day, a week, or a month, whereas the cost per hour ratio of a VM decreases with increasing subscription times. However, as decribed later in detail, an application owner needs to trade off these decreased costs per hour with a decrease in agility regarding the de-allocation of a VM.

In contrast to dedicated hosting models, VMs can be provisioned on-demand and can be sized according to an application owner's choice from a set of predefined VM sizes. However, the alignment of VM capacities to an application's changing resource demand in order to provide *enough resources* without wasting costly VM capacity is a challenging task. This task currently needs to be done manually by application owners.

Fig. 1 gives an example of a respective decision problem. The figure illustrates an application's CPU workload profile over time. Here, an application owner needs to decide when and which VM of offered sizes (A - C, whereas for example A_1, A_2 represent two VM instances of type A) should be allocated or de-allocated to a load balanced application in order to satisfy its varying CPU demand. Allocation decision have to be derived, considering that additional VMs can be allocated anytime (scale-out), but will be charged by minumum subscription times. For example, one has to purchase an instance of type A for a minimum duration of one hour or multiples. Hence, de-allocation (scale-in) of a VM is allowed only at certain points in time (for example, every three hours for VMs of type A and B, or every 12 hours for instances of type C). Here, we assume that the demand cycles during the day are known, which is regularly the case and a valid assumption for many enterprise applications as shown by Gmach et al. [5].

On the one hand, poor planning might lead to costly over-capacity, or SLA violations in case of insufficient capacity. On the other hand, automated capacity management requires continuous monitoring, planning and optimization to make adequate sizing and scaling decisions over time. The problem demands for an algorithmic and automated solution, since

[1]http://aws.amazon.com/ec2/
[2]http://www.gogrid.com
[3]http://www.slicehost.com

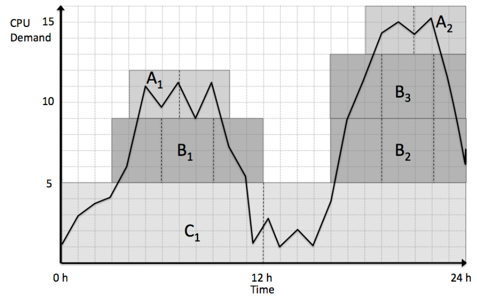

Fig. 1. Resource scheduling problem

manual capacity is likely to be suboptimal.

Therefore, we propose an automated decision making and enactment service to ensure the provisioning of sufficient VM resources for SLA compliant, but cost-minimal capacity over time. Capacity management is achieved by allocating or de-allocating VMs of predefined fixed capacities. Therefore, the cost minimization determines an optimal schedule for allocating and de-allocating new VM instances considering minimal subscription times and forecasted application demand.

In the following sections, we will propose decision models and a high-level architecture for an automated capacity management service. We aim at supporting application owners during the operational phase of an application life-cycle by automatically balancing out variable hosting costs against fluctuating resource demands. In addition, the service can be used to select IaaS providers based on their offerings (VM capacities and prices offered) and historical application demand.

In summary, the problem addressed in this paper can be stated as: *Given a number of unlimited, typed sets of VMs and an application with varying resource demands, we want to determine the cost-minimal schedule of VMs over time that satisfies the demand at each point in time, while minimizing the cost.*

II. Scaling Decision Model

In the following subsections we present a mathematical model to derive a cost-minimal schedule, followed by extensions for conditions and requirements often found in practice, and an algorithm to solve the problem.

A. Model Formulation

In our model we assume predictable resource demand throughout our planning period. Due to the *on-demand VM allocation option*, we can handle unexpected demand peaks during a planning period by allocating additional VMs instantly as required. Furthermore, in this short paper, we concentrate on a single resource (e.g., CPU) and assume this to be the primary bottleneck. While this is a justifiable assumption for many enterprise applications, the extension to multiple resources is straight forward.

Reminiscing Fig. 1, a VM allocation forms a rectangular geometrical shape with the size of a VM as the height, and the duration of the allocation as length. Consequently, this problem can be understood as a variant of the rectangle packing problem as discussed by [6]: How to pack a set of non-overlapping price-differentiated paraxial rectangles in a way that covers an area below a demand curve completely over a planning period with minimum total costs. Costs associated with a rectangle are the VM prices (per time slot) multiplied by the length of the rectangle. In general, the price per time slot decreases with increasing subscription time of a VM type (comparable to volume discount models often found in wholesale trade).

We divide time into discrete time slots t (t *=1, . . . , T*), wherein we assume a fixed demand level u_t (e.g., the maximum demand expected during an hour of a day). Let j (j *=1, . . . , J*) be a VM type with a maximum size s_j of a server resource (e.g. CPU performance or main memory). Let further Δt_j denote the indivisable minimal subscription time of an j-type VM. A VM allocated by a customer is always paid for this minimal subscription time or multiples (e.g., one or more hours). Hence, although a VM instance can be allocated anytime, de-allocating a VM is only possible at certain points in time. Let further c_j be the price to pay per time slot, as long as a j type VM is allocated.

Rectangle packing is an NP-hard optimization problem. While some NP-hard problems can be solved for practical problem sizes, rectancle packing cannot be expected to scale for any realistic number of time slots and VM types [6]. There are two possible remedies: Either, we simplify the problem, or we solve the full problem heuristically. In our paper, we will suggest both. In this subsection, we will propose an optimization model assuming minimal subscription times of equal length, while in the next subsection, we will provide a heuristic for those cases, where the minimal subscription times vary considerably and it is necessary to solve a modified rectancle packing problem.

In the following, we will set time slots t to the minimum subscription times Δt_j of VMs. As resource demand we take the maximum demand over an hour or a certain percentile of the resource demand. The demand can readily be estimated from historical load data [5]. This way, we can find an optimal set of VM instances for each time slot. The decision variable x_j describes how many instances of a VM type j are allocated in time slot t:

$$
\begin{array}{llll}
\min \sum_j c_j \cdot x_j & & & \\
s.t. & & & \\
\sum_j s_j x_j & \geq & u_t & \forall t \\
x_j & \in & \{0,1,...,W\} & \forall j \in \{1,...,J\}
\end{array}
$$

Notice that because the weight of each item is at least 1, we can never choose an item more than W times. While the objective function minimizes the total costs over all time slots, the first constraint guarantees sufficient capacity over the planning period as the application's resource demand must never exceed the total allocated capacity of all VMs. Note that we assume that demand can be efficiently balanced to allocated VMs. Risk averse planners can increase u_t to make sure that the available capacity is available in each time slot. Obviously, also the Basic Model is an integer programming problem. However, due to the fact that we solve the problem for each time slot, the problem size is so small that we can solve it in a matter of seconds or milli-seconds for realistic problem sizes.

There is usually a limit L regarding the maximum cluster-size, i.e., the number of concurrent VM instances of an application (for example, the cluster size of an Web application using in-memory session replication is only efficient for a limited number of cluster nodes). We address such a limitation by restricting the number of running instances per time slot to L:

$$\sum_j x_j \quad \leq \quad L \qquad \text{(Scale-Out Limit)}$$

If scale-out is not possible at all, resource demand cannot be balanced over VMs and we set $L = 1$. In a similar manner we can address further constraints like high availability requirements or total budget constraints.

B. Heuristic for the Problem with Heterogeneous Subscription Times

The basic model assumes equal subscription times for all VM types of one hour. However, some providers offer VMs with a longer subscription time (e.g., a day or a week), whereas the cost-per-hour ratio of a VM decreases with increasing subscription times. We expect to see even more of this type of volume discounts in the future. Obviously, decreased costs per hour need to be traded off against a decease in agility regarding the de-allocation of a VM. As indicated in the previous subsection, solving the VM scheduling problem optimally will then turn into a notorious NP-hard problem, which will not be tractable for larger problem instances. Consequently, in this subsection, we propose a heuristic solution for this problem.

For clarity, in the following we use superscripts $n, (n = 1, ...N)$ on VM types j to denote an order regarding their subscription times. For example, j^1 is a VM type with an subscription time of an hour, while $j^3 (= j^N)$ is a VM type with an subscription time of a week (in this case the maximum considered subscription time). As found in practice, we further assume sizes of $j^{n>1}$ to be multiples of the smallest j^1 and that prices for VMs with the same subscription time increase linearly with their sizes. Hence it suffices to consider the smallest VM type per subscription time as later one can still decide for example to choose a VM of the double size instead of two VMs of the same type because of cluster-size limits. Furthermore, we can always substitute the demand covered by a VM by sets of VMs with shorter subscription times.

We now propose a recursive algorithm to decide whether to allocate VMs with longer subscription times. We set the duration of the planning period to the longest subscription time of offered VMs. The basic idea of our heuristic is to cover as much demand as possible with VMs with long subscription times since their unit capacity costs are the lowest. Once longer subscription times become more expensive for the remaining demand (because of increasing over-provisioning due to coarse grained resource allocation), VMs with smaller subscription time are used until we reach an hourly subscription basis. We then revert to our decision model as described in the previous subsection. The pseudo-code for our algorithm is as follows:

```
Algorithm: VM Scheduling
1 $n := N$
2 $while(n \geq 2)$
3   $while(\forall t : u_t \geq s_{j^n})$
4     $allocate(j^n)$
5     $demand := demand - s_{j^n}$
6   $while(alternative(j^n, j^{n-1}...j^1,\ demand)$
7   $n := n - 1$
8 $basicModel(demand)$
```

Starting with the VM type with the maximum subscription time (*line 1*), we repeat the loop starting in *line 2* for VM types with decreasing subscription time until we reach the VM type with minimum subscription time ($n = 1$). In *line 3* we start an inner-loop repeated as long as the size of a VM j^n is fully required throughout its subscription time, i.e., resource demand always exceeds s_{j^n}. Then, a j^n instance is allocated (*line 4)* and we repeat this inner-loop with the remaining resource demand not covered by the j^n instance (*line 5)*.

We exit this inner-loop once the remaining resource demand partly falls below the size of j^n. In this case we need to decide whether it is still beneficial to allocate j^n as on the one hand unused capacity of j^n needs to be paid, on the other hand its cost per time slot is lower than those of alternatively allocating a series of VMs with shorter subscription times and better demand curve alignment. Function *alternative()* in *line 6* (described in the subsequent paragraphs) decides whether costs are lower when allocating j^n or if it is cheaper to cover the demand by alternatively allocating a set of VMs with shorter subscription times. As long as j^n is the cheaper alternative a j^n instance is allocated and function *alternative()* is re-executed with the remaining resource demand. Otherwise the iteration of the outer-loop ends and the next iteration is started with j^{n-1} until we reach $n = 1$. After the main loop stops, we solve our basic model introduced in the previous subsection - parameterized with the remaining resource demand to determine optimal allocations for the VMs with minimum (hourly) subscription times.

Function *alternative()* uses the following heuristic to make a VM j^n type allocation decison. This decision clearly depends on how much overall demand it covers. By measuring resource demand in units per time slot (subscription time of j^1) we determine the sum of the overall resource demand covered by j^n over its subscription time. By dividing its price by this sum we obtain j^n's *effective price* per demand unit covered and we allocate j^n if and only if its *effective price* does not exceed the average price per demand unit if the resource demand would be covered by *alternatively* allocating combinations of multiple VMs of shorter subscription times $(j^{n-1}, ..., j^1)$.

The *alternative price* is determined in a recursive manner: the demand over the subscription time of j^n is partitioned into a sequence of durations matching the length of the subscription time of j^{n-1} (for example, if subscription time of j^n is one week, and the subscription time of j^{n-1} is one day, we get seven one-day-partitions). For each partition, *VM-Allocation* is called, initialized with $n = n - 1$ (in line 1) and the resource demand that would be covered by j^n (in this partition). The *alternative price* is then computed as the sum of the costs of VM allocations over all partitions.

III. Solution architecture

To implement and execute the model decisions we propose a control architecture as depicted in Fig. 2

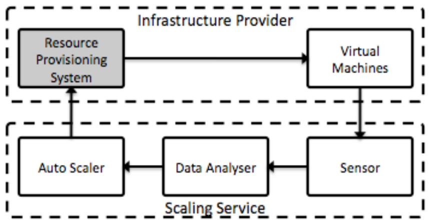

Fig. 2. Control architecture

Sensors are used to acquire resource usage metrics of the various VMs. Therefore, standard server monitoring-tools are suitable. The monitoring data is send to a data analyzer for trace-based prediction of workload throughout a planning period. The auto scaler derives scale-in/out and/or scale-up/down decisions based on the predicted workload for different server resources, the available VM sizes and their prices, subscription times and maybe resource provisioning delays (time required to set-up a VM). The auto scaler is comparable to an autonomic manager that tunes, load balances and derives scaling decisions based on a predictive system model following the control system approach [7]. The resource provisioning system allows to abstract from technical specifica of cloud infrastructure providers. It is in charge of executing these commands in a timely manner independent of the technologies and protocols used by specific providers.

Typically business applications consist of multiple components like load balancers, web, application and database servers. However, the proposed model can be applied by considering each tier as single application as we need to make sure that capacity suffices on each tier to guarantee SLA compliant application delivery.

IV. Summary and Outlook

Infrastructure-as-a-service providers offer different types of virtual machines, this means different capacities, prices, and minimum subscription times. Customers typically have applications with volatile demand throughout the day or week. They want to purchase as much capacity as necessary, in order to minimize cost, while satisfying resource demands of an application. It turns out, that determining an optimal schedule of VM subscriptions is a hard computational problem. In this paper, we propose different problem formulations and algorithmic solutions.

Solving this problem allows to select the lowest-cost IaaS provider. The solution can also be used to automate the capacity management and control the schedule of VM instances that need to be allocated over time. Such automated capacity management can considerably lower the operational costs when relying on external IaaS providers. We are currently implementing a software solution and are working on a solid experimental evaluation of the approach.

References

[1] J. Rolia, L. Cherkasova, M. Arlitt, and A. Andrzejak, "A capacity management service for resource pools," in *5th Intl. Workshop on Software and Performance (WOSP'05)*, 2005.

[2] M. Arlitt, D. Krishnamurthy, and J. Rolia, "Characterizing the scalability of a large web-based shopping system," *ACM Transactions on Internet Technology*, vol. 1, 2001.

[3] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal, "Dynamic provisioning of multi-tier internet applications," in *In Proceedings of the 2nd International Conference on Autonomic Computing*, 2005, pp. 217–228.

[4] A. Chanda and P. Shenoy, "Effectiveness of dynamic resource allocation for handling internet flash crowds," University of Massechusetts, Amherst, Tech. Rep., 2003.

[5] D. Gmach, S. Krompass, A. Scholz, M. Wimmer, and A. Kemper, "Adaptive quality of service management for enterprise services," *ACM Transactions on the Web (TWEB)*, vol. 2, 2008.

[6] J. E. Beasley, "Bounds for two-dimensional cutting," *Journal of the Operational Research Society*, vol. 36, no. 1, pp. 71 – 74, 1985.

[7] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury, *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.