From Data Center Resource Allocation to Control Theory and Back

Xavier Dutreilh and Nicolas Rivierre
Orange Labs, Issy-Les-Moulineaux, France
Email: {xavier.dutreilh,nicolas.rivierre}
@orange-ftgroup.com

Aurélien Moreau and Jacques Malenfant
Université Pierre et Marie Curie - Paris 6 CNRS,
UMR 7606 LIP6, Paris, France
Email: {Aurelien.Moreau, Jacques.Malenfant}@lip6.fr

Isis Truck
Université Paris 8,
EA 4383 LIASD, Saint-Denis, France
Email: truck@ai.univ-paris8.fr

***Abstract*—Continuously adjusting the horizontal scaling of applications hosted by data centers appears as a good candidate to automatic control approaches allocating resources in closed-loop given their current workload. Despite several attempts, real applications of these techniques in cloud computing infrastructures face some difficulties. Some of them essentially turn back to the core concepts of automatic control: controllability, inertia of the controlled system, gain and stability. In this paper, considering our recent work to build a management framework dedicated to automatic resource allocation in virtualized applications, we attempt to identify from experiments the sources of instabilities in the controlled systems. As examples, we analyze two types of policies: threshold-based and reinforcement learning techniques to dynamically scale resources. The experiments show that both approaches are tricky and that trying to implement a controller without looking at the way the controlled system reacts to actions, both in time and in amplitude, is doomed to fail. We discuss both lessons learned from the experiments in terms of simple yet key points to build good resource management policies, and longer term issues on which we are currently working to manage contracts and reinforcement learning efficiently in cloud controllers.**

***Keywords*-Cloud computing, Application hosting, Resource allocation, Closed loop systems, Controllability, Hysteresis.**

I. Introduction

Cloud computing fosters a very agile service-based computing market by providing an easy access to computing resources in order to rapidly deploy new services with full access to the World-Wide Web market. Indeed, for a service provider, having an almost instantaneous access to virtualized computing resources offered by clouds reduces drastically the time-to-market, a key issue in such a business.

Compared with traditional means to provide computing resources for applications, namely for service providers to install their own physical servers and connect them securely and safely to the web, cloud computing promises an easier access to resources and the web, but also allows for allocating resources as leanly as possible. Given the high variation in workload of Internet applications, a lot of work currently focus on making clouds actively revise resource allocations to applications as their workload changes. Such revisions indeed serve two purposes: keeping the costs as low as possible for service providers, and allowing for resource sharing between applications for the clouds, a necessity for the overall optimization of resources especially regarding energy consumption and green computing.

Closely linked to the field of autonomic computing, several works try to implement in clouds such resource management policies. The most popular policies to date are threshold-based policies, where a performance target determined in negotiated SLA is flanked by lower and upper thresholds triggering allocation and deallocation of resources to the application. Some authors have also compared this resource allocation problem to an automatic control one, and therefore apply known techniques such as reinforcement learning to learn policies as the applications run.

In this paper, we analyze both approaches from experiments we have made in the context of a private cloud. These experiments show that setting thresholds manually and computing policies through reinforcement learning techniques are both tricky, and in fact doomed to fail if not implemented with much care. Setting thresholds need a deep understanding of well-known phenomena in automatic control, such as instability. Applying reinforcement learning requires a full-fledged integration methodology taking care of pragmatic issues, such as initialization, good choice of policy exploration strategies and convergence speed-ups, without which the overall control will fail.

The rest of the paper is organized as follows. Section II sets the scene for automatic resource allocation in clouds. Section III presents VirtRL, the framework used in our experiments. Section IV details the protocols and results of the experiments we made with the different control policies. Section V aims at learning lessons from the experiments in form of advices to cloud managers when implementing resource allocation policies. A conclusion then follows.

978-0-7695-4130-3/10 $26.00 © 2010 IEEE
DOI 10.1109/CLOUD.2010.55

IEEE computer society

II. Resource allocation in data centers

A. Motivation

Cloud computing is the delivery of resources and services on an on-demand basis over a network. Three markets are associated to it. Infrastructure-as-a-Service (IaaS) designates the provision of IT and network resources such as processing, storage and bandwidth as well as management middleware. Platform-as-a-Service (PaaS) designates programming environments and tools supported by cloud providers that can be used by consumers to build and deploy applications onto the cloud infrastructure. Software-as-a-Service (SaaS) designates hosted vendor applications. IaaS, PaaS and SaaS all include self-service (APIs) and a pay-as-you-go billing model. Companies can use clouds either to run punctual batch jobs (e.g. video transcoding) or web applications. The cloud is appealing to them because of its ability to reduce capital expenditures and increase return on investment (ROI) since the traditional model where physical servers were bought and amortized in the long term is no more.

Many web applications face large, fluctuating loads. In predictable situations (new campaign or seasonal), resources can be provisioned in advance through proven capacity planning techniques. But for unplanned spike loads, auto-scaling is a way to automatically adjust resources allocated to an application based on its needs at any given time. The core features are (i) a pool of available resources that can be pulled or released on-demand, and (ii) a control loop to monitor the system and decide in real time whether it needs to grow or shrink. Auto-scaling is offered in PaaS environments by providers like Google App Engine [1] but applications are developed specifically for these platforms. IaaS is more flexible since users are given free access to virtualized hardware, relying on providers like Amazon [2] or open-source projects like Eucalyptus [3] and OpenNebula [4] to instantiate VMs (taking as input a set of resource attributes: CPU, memory or storage). IaaS issues however are the lack of (i) widely adopted standards, although initiatives such as DMTF OVF and OGF OCCI are active toward the definition of virtualization formats and IaaS APIs, and (ii) automation since developers must build the machinery, or use third party tools such as RightScale [5]. This paper focuses primarily on resource allocation policies that could be used in IaaS management layer to perform auto-scaling.

B. Resource allocation and policies

Fostered by autonomic computing concepts, allocating resources to applications in clouds has been the subject of several works in recent years. In this paper, we concentrate on two types of policies:

1) threshold-based policies, where upper and lower bounds on the performance trigger adaptations, where some amount of resources are allocated or deallocated (typically one VM at a time);
2) sequential decision policies based on Markovian decision processes (MDP) models and computed using, for example, reinforcement learning.

Threshold-based policies are very popular among on-the-field cloud managers. The simplicity and intuitive nature of these policies make them very appealing. However, as our experiments show, setting thresholds is a per-application task, and can be very tricky. As an alternative to manual threshold-based policies, modeling the system as a MDP allows for computing policies that can take into account the inertia of the system, such as fixed costs to allocate or deallocate virtual machines, which favors retaining the same number of virtual machines when the variation in the workload does not last enough time to amortize these fixed costs. Such compromises are the cornerstones of sequential decision making.

C. Control theory concepts

The basic ingredients of a control system are: (i) objectives of controls, or inputs, (ii) control-system components and (iii) results or outputs [6]. In closed-loop control, part of the inputs are state variables providing information on the controlled system that can be used by the control-system components to decide upon the value of the correction to apply to get the desired output.

Applying feedback to control has been the target of the broad discipline of automatic control since at least WWII. Feedback can of course help stabilizing an otherwise uncontrolled system, but can be equally harmful if not properly used. Broadly speaking, stability characterizes the pace of variation in the output of a closed-loop control system. When the variation is too high, the arising instability may have adverse impacts on the control system. In cloud systems, an unstable resource allocation policy can result in a form of thrashing, where the system devotes more time to allocating and deallocating resources than to the execution of applications, therefore impairing its overall performance.

Automatic control shows us that there are three main sources of instability in control systems:

- **latency** to get the stationary values of observable variables after applying a control action;
- **power** of the control, when the minimal magnitude of the control has a too large impact on the magnitude of the outputs, making them immediately out of the control objectives;
- **oscillations** in the inputs, when they are too large to be smoothed out by a too weak control, therefore appear almost *as is* in the outputs.

D. Related work

Several works apply resource allocation techniques in cloud computing. We concentrate here on the ones that take a disciplined rather than an ad hoc approach to the problem.

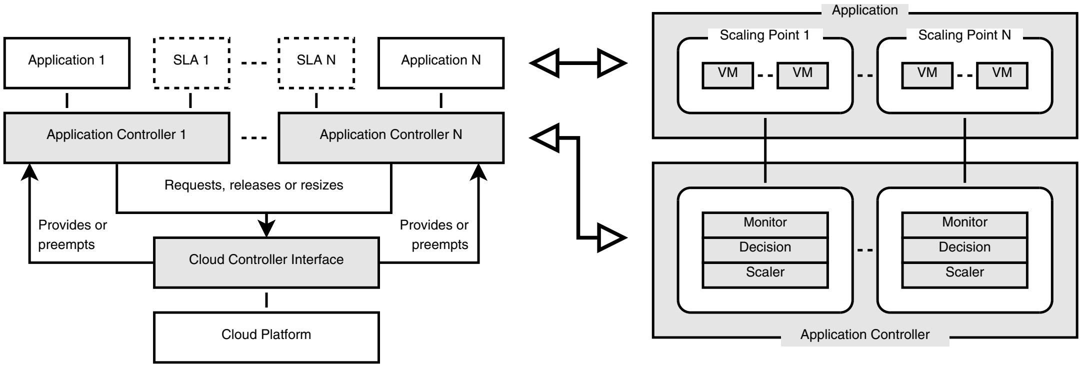

Figure 1. The VirtRL framework architecture

Among the different works on threshold-based policies, Lim et al. [7] propose to narrow the range between thresholds when more frequent decisions are needed, e.g. coping with flash crowds. Their proposal fits well into the framework proposed here to adjust the control to the current conditions.

Xu et al. [8] construct their controller in two levels. A first level applies fuzzy logic techniques to learn the relationship between workload, resources and performance. It then controls the resource allocation by deciding at a regular interval the level of resources needed by each application, thanks to the fuzzy rules previously learned. The second level applies a simply knapsack technique to allocate the resources to the different applications. Though interesting to learn the behavior of applications, this approach limits itself to homogeneous applications by computing only one set of rules, and does not really care about stability in its control related part. Rao et al. with their VCONF [9] apply reinforcement learning but in the context of neural networks, in a way that parallels the work of Xu et al. Their neural network represents the relationship between workload, resources and performance used to perform vertical scaling.

Tesauro et al. [10] explore the application of reinforcement learning in a sequential decision process. The paper presents two novel ideas: the use of a predetermined policy for the initial period of the learning and the use of an approximation of the Q-function as a neural network. The results are interesting, though dependent on the form of the reward function. Besides that, the initial learning with a predetermined policy appears less promising than an initialization using a precomputing of the Q-function through the traditional value-iteration algorithms in a model-based learning approach [11].

Zhang et al. [12] propose a pragmatic approach to resource allocation, which consists in preallocating enough resources to match up to 95% of the observed workload, and then allocates more resources on another cloud when this threshold is passed. No real automatic control approach is applied to prevent instability, however.

Kalyvianaki et al. [13] apply statistical and Kalmann filtering approaches but limit themselves to vertical scaling within the bounds of one physical server.

III. VirtRL Architecture

In response to the strong need for autonomic resource allocation in cloud computing, we introduce VirtRL, a management framework dedicated to automatic resource allocation in virtualized applications hosted on cloud platforms such as Amazon EC2 [2]. Our prototype has been tested on OpenNebula [4] and on a private IaaS solution developed at Orange Labs. VirtRL relies on a two-level architecture and defines two roles: the cloud provider and the service provider. As stated in [8] and [7], we think a decoupled control in cloud computing is a must-do. The cloud provider operates its infrastructure to meet its own business objectives (e.g. workload placement and arbitration) while the local control of each application can be left to the service provider. Resource allocation can be driven by application-level SLA goals (average response time or session count for instance) and the current workload (e.g. request rate). Figure 1 depicts the VirtRL architecture.

The cloud controller interface abstracts IaaS platforms. It publishes a list of virtual machine descriptions supported by the underlying platform, and then offers three generic non-blocking services (VM allocation, resizement, destruction) and two asynchronous services (resource delivery, and preemption to get back resources when required). Finally, its design allows the use of resource sharing policies such as arbitration [8] or green optimization [14], [15].

The application controller allows to automatically adjust the capacity of virtualized applications, given performance goals and current workload. Its design is open and allows to attach custom on-line scaling policies to one or several potential bottlenecks (front end, application layer, etc) of an application, using the concept of *scaling point*. Each scaling point represents a resizable group of virtual machines controlled by a control-loop composed of three submodules: monitor, decision policy and scaler. The monitor keeps track

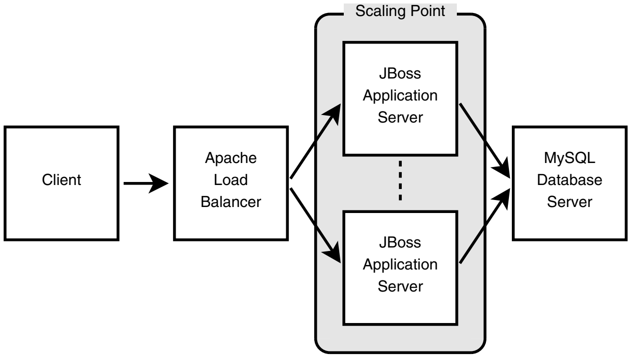

Figure 2. RUBiS architecture and its scaling point

of the current state (workload, performance and resource usage). At specific time, the decision policy comes into action and decides if some action should be taken: grow or shrink the resources in the scaling point, or do nothing. If the chosen action modifies the VM count of the associated scaling point, the corresponding operations (VM allocations and destructions) are executed on the cloud controller through its interface. Finally, the scaler is in charge of the initialization of all virtual machines before registering them into a scaling point. In case of a removal, it ensures that the server is correctly unregistered from the application before releasing its resources.

IV. Experiments

We now attempt to identify from experiments and the knowledge of the cloud and of the applications, the sources of instabilities discussed in section II-C, striving for simple yet efficient rules to obey in order for cloud managers to get good resource management policies.

A. Experimental Setup

Testbed. To experiment our control system, we use the Servlet edition of the RUBiS application [16], a prototype of an auction website. The application consists of three main components: Apache load balancer server, JBoss application server and MySQL database server. In the initial setup, each component runs in a separate virtual machine (VM). The same profile is used for all VMs during the experiments: one computing unit (equivalent to 2.66 GHz guaranteed), 256 MB of memory and 10 GB of storage.

Figure 2 gives an overview of the RUBiS architecture and the scaling point defined for our experiments. The VirtRL framework makes the RUBiS application tier a scaling point in order to evaluate two scaling policies (static threshold-based and Q-Learning-based) discussed below. The VirtRL monitoring API allows to collect data at constant time intervals (20 seconds here). Each data represents the state of the system during a time interval, and is composed of: average request rate (workload), average response time of RUBiS (performance), and number of VMs in the scaling point (resource usage). The VirtRL load balancer API (wrapping the Apache load balancer) allows the controller to scale up or down the application tier, depending of the decision taken by the scaling policy in use.

Workload pattern. For the sake of our experiments, an edited version of the RUBiS load injector is used. We use a flash crowd workload pattern to highlight the impact of on-line scaling policies when stressed by the first source of instability discussed in section II-C (oscillation in the inputs). This pattern is made of five sinusoidal oscillations. All oscillations share the same period (an hour and half) but differ in their slope to simulate different maximum request rates. The smallest oscillation emulates a workload range which goes from zero to twelve requests per second while the strongest one goes from zero to sixty requests per second. To give enough training time to the reinforcement learning policy to capture the performance model and its management strategy, the pattern is repeated seven times, making the experiment lasting for more than sixty hours.

B. Static threshold-based policies

As in Amazon Auto-Scaling [17] and Rightscale [5], static threshold-based policies can be used to dynamically adapt the resources of virtualized applications. Our implementation relies on SLA metrics (as average response time) to meet performance objectives (SLOs). The policy has five parameters: an upper threshold ut, a lower threshold lt, a fixed amount of virtual machines n to be allocated or deallocated, and two inertia durations: $inUp$ for scaling up and $inDown$ for scaling down. When carefully choosen, these parameters hide the latency phenomenon (see section II-C), in allocating/deallocating VMs, initializing JBoss servers and rebalancing equally client requests among existing servers (after a resource addition or a removal). At each iteration, if the performance violates ut, n VMs are requested and registered into the scaling point. Then, the controller inhibits itself for $inUp$ seconds. If the performance goes below lt, n VMs are removed and their resources released. Again, the controller inhibits itself for $inDown$ seconds. In our experiment, we set the following values: $ut = 0.8$, $lt = 0.4$, $n = 1$, $inUp = 120$ and $inDown = 60$. The SLO is: “the average end-user response time must not be higher than ut.”

Figure 3 illustrates the results of our first experiment. Several phenomena can be explained. The first one is about the tricky policy parameter setup. In the first place, a service provider has to bench by himself the target system to capture its performance model. Then, given a high-level objective to achieve under a specific workload, he can choose the correct parameters to meet this SLO. On the load pattern, the second and the third sinusoidal oscillations show a case where parameters are well-suited. When the workload increases and current virtual servers become overloaded, SLA violations occur but do not last since the policy is configured to scale up quickly.

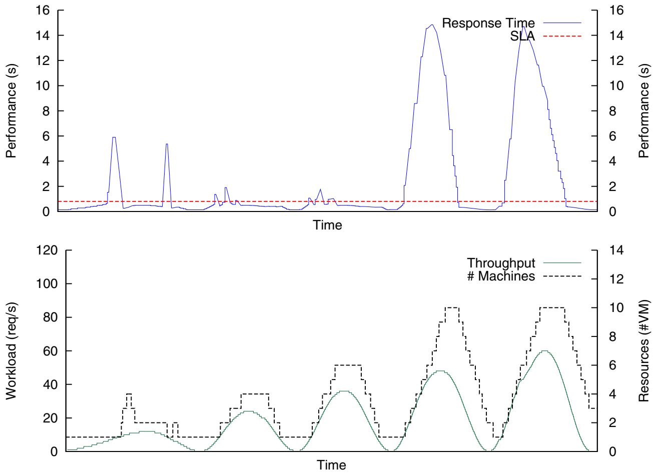

Figure 3. Result of a 7.5 hours experiment with the static threshold-based policy. The top graph shows the SLA objective and the average response time evolution. The bottom graph shows the workload and the VM usage. The figure shows that the policy configuration is well-suited for the second and the third oscillations.

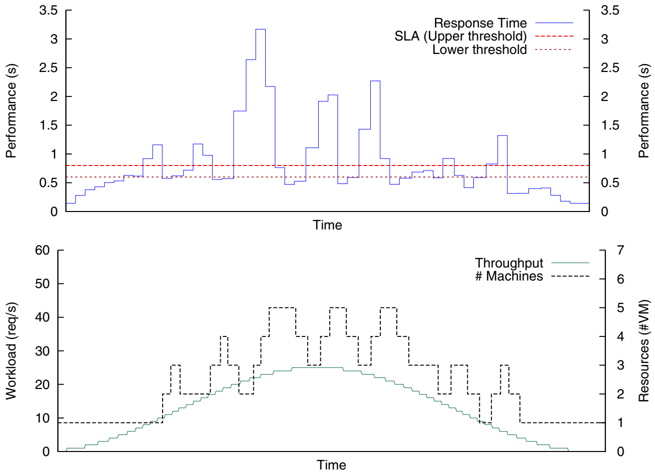

Figure 4. Result of the second oscillation replayed with closer thresholds: $ut = 0.8$ (SLO) and $lt = 0.6$. The policy constantly allocates and deallocates VMs. Because of the latency in these actions, SLA violations occur which cause service unavailability.

Nevertheless, when the control power and the slope differ too significantly, two sources of instability appear. The first oscillation shows what happens when the control is stronger than the slope. Unnecessary servers are allocated and then released at the next iteration. Being too reactive makes the system fooled in its performance and causes instability. On the other hand, when the control is weaker than the slope, some of these characteristics are also inherited. On the fourth and the fifth oscillations, the decisions are taken too lately which causes another form of instability, and actually service unavailability. Because the strength of the control (the parameter n) is chosen once and for all, current threshold-based policies are particularly subject to these kinds of instability. During these periods, the reported performance is based on the response time of served requests and rejected ones are not taken into account.

Finally, Figure 4 shows what happens when the latency is not properly hidden from the decision module. We replayed the second oscillation with a policy using some closer thresholds. This setup demonstrates the unfeasibility of capturing the latency phenomenon. When violations occur, a new VM is requested and added to the scaling point. Then, as soon as the performance increases, the policy finds the last created VM useless due to the close lower threshold. So, it tends to release it even if it causes future SLA violations.

C. *Q-Learning-based policy*

Instead of relying on static thresholds and amount of VMs to allocate or deallocate, the Q-Learning approach [18] captures online the performance model of a target application and its policy without any *a priori* knowledge. At time t, given a state s_t (current workload and resource usage), it looks for the best action a_t to execute. Possible actions are: (i) add n new VMs, (ii) release n VMs, or (iii) do nothing. Since they have consequences on the target application over the time, a scalar measurement called reward is calculated at time $(t+1)$ to evaluate the goodness and the badness of a particular a_t. In our implementation, a high reward is returned when SLOs are met (minus a certain penalty if the scaling point is over-provisioned) and a negative one otherwise. Therefore, and unlike our threshold policy, Q-Learning captures this history information into a value table. This one maps all system states s_t to their best action a_t and can be initialized with chosen values or based on a specific but simple performance model. Then, a training phase is required to capture the real model before converging to the optimal policy. The formula used in our Q-Learning implementation is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t)(1 - \alpha_t(s_t, a_t)) + \\ \alpha_t(s_t, a_t)\,(r(s_{t+1}, a_t) + \gamma \max_a Q(s_{t+1}, a))$$

Here (s_t, a_t) are the state and action at time t, $r(s_{t+1}, a_t)$ is the delayed reward for a_t evaluated at time $(t+1)$, $\max_a Q((s_{t+1}, a)$ denotes the Q value of the best action into the next state at time $(t+1)$, the constant $\gamma \in [0, 1]$ is a “discount parameter” expressing the present value of expected future reward and α_t is a “learning rate” parameter defining the impact of the new data on the Q function.

Apart from the reward function, four parameters can be customized: learning rate α, discount factor γ and two inertia durations: $inUp$ for scaling up, $inDown$ for scaling down. Both durations hide the latency phenomenon, as in the threshold case. In this experiment, we set the following values: $\alpha = 0.05$, $\gamma = 0.5$, $inUp = 60$ and $inDown = 60$. Beware the SLO remains the same as before and is explicitly expressed through the reward function.

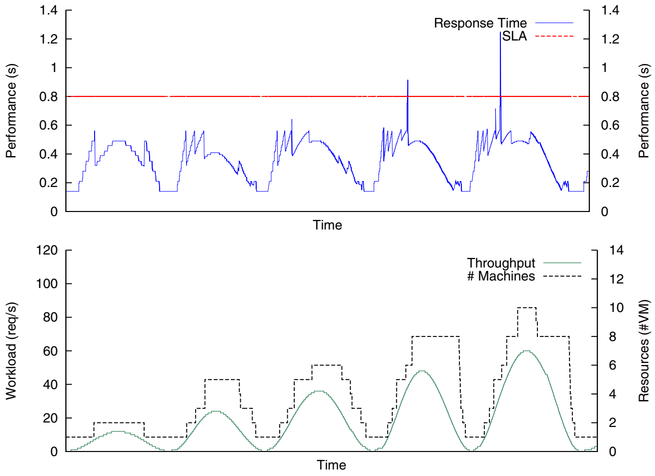

Figure 5. Result of the Q-Learning policy. Notice that the two graphs do not cover the training phase. The figure shows that once the performance model has been learned, the two sources of instability (power of the control and oscillations in the inputs) tend to disappear since the policy adapts its power to the oscillation slopes.

At first, the training phase was drastically long. As most of the works in this area, we were using a pure random trial-and-error method to capture the performance model. In production, such approach is almost unfeasible because it tests all combinations of states and actions. To overcome these annoyances, we used a policy based on a heuristic inspired from [19] during the training phase. The heuristic consists of pushing the system close to its limits by removing VMs one by one until SLA violations occur. Then, it immediately allocates a random amount of new VMs. This heuristic tends to capture faster a performance model, but limits the state space exploration.

Figure 5 illustrates the results of this second experiment after the training phase. Several phenomena can then be explained. It first shows that, with a well-captured performance model, SLA violations are kept to a minimum. Given a state s_t, the Q-Learning policy converges to the number of VMs to be allocated or deallocated. With various sinusoidal oscillations, it helps to adapt the control power to the workload slopes. Then, the two sources of instability (power and oscillation in the inputs) tend to be solved. Nevertheless, this heuristic is not flawless. The initial policy has the effect of dividing the training period into an initialization period and a learning one. The time spent in the initialization period is extremely reduced but still there. In this second experiment, it lasts for more than sixty hours. In the field of dynamically scaling applications, such times can be seen as irrelevant when a static threshold-based policy is operational in a few minutes. Moreover, as this first period skips a lot of states, Q-Learning needs the second real learning period, exploring more states to look for optimal actions to execute.

Finally, even if such actions are captured, the latency phenomenom still remains and it is partially induced by the underlying cloud platform. Hence, improving the target policy reactivity and, in the mean time, reducing its training period duration require to speed up the whole VM provisioning process. To this end, advanced system mechanisms such as pools of hot spare machines [19] or VMs with copy-on-write file systems should be introduced as a complementary-but-necessary step for reactive auto-scaling policies.

V. Lessons learned

We now summarize the lessons learned from the previous experiments on approaches to resource allocation in clouds.

A. *Resource allocation as automatic control*

As we have seen, continuously adapting the level of resources provided to applications to match their performance requirements is indeed an automatic control problem, and as such is subject to well-known problems that can be avoided or at least mitigated by observing the following "*from-the-field*" lessons:

1) For all the possible adaptation actions, measure the patterns of evolution of the performance against the time since the start of the adaptation action. This allows to capture the relationship between the amplitude of the return in performance and the time required for the application to stabilize the controlled to this new performance level, and the set or the amplitude of actions. The knowledge of the relationship is crucial to understand the behavior of the controller.
2) Make sure that your control system does not try to make adaptations at a faster rhythm than the time required to stabilize the performance of the system, otherwise the instability will occur. As we have seen, when a new action is made before the performance has stabilized, the wrong performance level may fool the controller and trigger an allocation or deallocation which is not needed. In this case, the performance will stabilize itself at a too high or too low level, immediately triggering inverse decisions, thus entering in an infinite loop that will make the system unstable.
3) Analyze the workload patterns of variation to which the application needs to adapt, and make sure that: (i) the time required to stabilize to the new performance levels after adaptation actions is small enough to cope with these patterns; (ii) the maximum performance return given the possible adaptation actions is high enough to cope with the maximum expected variation in workload between two adaptation decisions.

On the third point, indeed, given the slope of the workload variation, two parameters of the control algorithm must be set: the power of the control actions and the time between two adaptation decisions. The time delay between two decisions must be long enough to let the performance stabilize itself to its new expected level. Furthermore, the amount of variation in the workload between two decisions is a direct consequence of this minimum time to stabilization, which in turn imposes a minimum level of power to the actions to cope with this amount of variation. For example, if the system needs thirty seconds to stabilize to its new performance after a VM allocation, the next decision cannot be made before these thirty seconds, and then the workload variation can be as large as the variation pattern authorizes within thirty seconds. If this variation is so large that the application needs two new VM, then the adaptation actions must allow for the allocation of these two VM.

In conclusion, as part of an automatic control problem, the mediation between the controller and the controlled system must obey several constraints. Trying to implement a controller without looking at the way the controlled system reacts to actions, both in time and in amplitude, is doomed to fail. This is of paramount importance in the context of cloud computing, where applications to be controlled comes from possibly untrusted third parties. Hence, besides the more traditional SLA, there is an urgent need for control contracts establishing, to summarize, that a controller can be held responsible for a good resource allocation if and only if the application fulfills corresponding commitments in terms of converting amounts of allocated resources into better performance within known delays.

B. Finding good resource allocation policies

We have studied two distinct types of control policies: threshold-based and sequential decision models.

In the first case, experiments have shown that manually setting the thresholds is tricky. An immediate temptation is to put tight thresholds seeking for an "optimal" resource allocation policy: to allocate or deallocate VM as soon as the performance moves as slightly as possible away from the target (SLO). But, for a given pattern of workload variation, the difference between the upper and the lower bounds determines the minimal time between two adaptation decisions. Hence, we have observed instability when setting thresholds too tightly in two cases: (i) when the workload variations are large enough so that successive crossings of one threshold or the other occur at a too high frequency compared to the required delay for the performance to stabilize; (ii) when the allocation or deallocation of a virtual machine, the least possible action when crossing a threshold, cause the performance to immediately cross the other threshold. In this case, we can recognize from an automatic control well known situation where the control is too powerful. On the other hand, for given maximal control decisions (e.g. if the policy is limited to allocating or deallocating virtual machines one at a time), a too long delay between decisions may result in having a too large amount of variation to cope with, impairing the controllability of the system. Hence, the difference between the two thresholds cannot be arbitrary large either. Lessons learned from these experiments can therefore summarize to:

1) For all possible adaptation actions, measure the new stabilized performance after crossing the lower (resp. upper) threshold, and make sure the upper (resp. lower) threshold is strictly less than (resp. greater than) this measure.
2) From the maximal time t_s to stabilize the performance after any adaptation action, compute the difference between the two thresholds so that the time to pass from one threshold to another is larger than t_s even for the maximal slope in the workload variation.

Regarding the alternative, i.e. modeling the system as a MDP, reinforcement learning is, in theory, a good candidate for solving such models by finding (near-)optimal decision policies. Compared to more traditional operation research techniques, reinforcement learning need not to establish models of state transitions and rewards *a priori*, but rather learn these as well as the optimal policy while running the system. An immediate benefit of this approach is therefore to be able to deal with new applications on-the-fly, a mandatory requirement in the cloud computing scenario. However, as noted in the related work [10], tuning reinforcement learning, albeit quite a pragmatic concern, is essential to get well-behaved systems in all circumstances. Besides defining a good Markovian decision process model, several lessons have been learned from our experiments:

1) Reinforcement learning needs an exploration of the decision space in order to gather information about the system behavior, but as noted by other authors [10], this exploration policy cannot be fully random, as some decisions may put the system in adverse states to which cloud clients shall never be exposed. Hence, make sure the exploration policy balance correctly the need for conservative decisions to satisfy clients and the need for speculative decisions to allow for discovering the best possible control policy.
2) Reinforcement learning requires a fair amount of observations to converge to an optimal, or simply good enough control policy. Albeit also quite pragmatic, any usage of such techniques requires a careful tuning of initializations and convergence speed-up techniques.
3) Reinforcement learning is based on the hypothesis that the system to be learned is fixed, i.e. the behavior of the system does not change over time. For applications having a very long execution time, this hypothesis is rarely true, as software updates and other hardware evolutions have an impact on this behavior [19]. When system evolution must be taken into account, several lessons can been learned as well: (i) estimate the rate of evolution of the system, and make sure that the time for reinforcement learning to converge to acceptable policies is smaller than the time for the

system to evolve to a new behavior; (ii) implement techniques (e.g. statistical testing) to detect changes in behavior of the system, and use them to trigger partial or full reinitialization of the reinforcement learning so to learn the new behavior as soon as possible.

From our experiments, we have concluded that MDP and reinforcement learning can be a good approach to solve the problem of resource allocation in clouds, but this is true only if it is put into an automatic workflow that deals with these pragmatic aspects: initializations of the learning, definition of an appropriate exploration policy for the learning, use of convergence acceleration techniques, and integration with other techniques to detect changes in behavior of the system to know when new policies must be learned.

VI. Conclusion

This paper is rooted in the observation that resource allocation in clouds, as an automatic control problem, is much harder than can be thought at first sight. From experiments illustrating adverse behaviors, such as instability in the allocations, link to well-known phenomena have been drawn and explained in terms of application behaviors. After this dive into automatic control concerns, the paper comes back to resource allocation in clouds by listing lessons learned in terms of a set of simple yet key points to measure, verify and set to build good resource management policies.

On a longer term, the paper identifies two promising concepts on which we are currently working:

1) developing an automatic and complete workflow to manage the implementation and the seamless integration of techniques such as reinforcement learning in the cloud control to provide for optimal resource management policies tailored to each application ;
2) defining the concept of control contract, that can be viewed as a SLA over traditional SLA, or meta-SLA, that would deals with the properties of the resource management required to implement the SLA, as well as the respective commitments of the parties for the control to be efficient.

Both of these points are crucial to the achievement of cloud computing, because of the use cases of this field which require to discover and install controls over third party applications on-the-fly.

References

[1] "Google App Engine," http://code.google.com/intl/fr/appengine/.

[2] "Amazon EC2," http://aws.amazon.com/ec2/.

[3] "Eucalyptus," http://open.eucalyptus.com/.

[4] "Opennebula," http://www.opennebula.org/start.

[5] "Rightscale," http://www.rightscale.com/.

[6] F. Golnaraghi and B. Kuo, *Automatic Control Systems*, 9th ed. Wiley, 2010.

[7] H. C. Lim, S. Babu, J. S. Chase, and S. S. Parekh, "Automated control in cloud computing: challenges and opportunities," in *ACDC '09: Proceedings of the 1st workshop on Automated control for datacenters and clouds*. ACM, 2009, pp. 13–18.

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

[9] J. Rao, X. Bu, C.-Z. Xu, L. Wang, and G. Yin, "Vconf: a reinforcement learning approach to virtual machines auto-configuration," in *ICAC '09: Proceedings of the 6th international conference on Autonomic computing*. ACM, 2009, pp. 137–146.

[10] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani, "A hybrid reinforcement learning approach to autonomic resource allocation," in *ICAC '06: Proceedings of the 2006 IEEE International Conference on Autonomic Computing*. IEEE Computer Society, 2006, pp. 65–73.

[11] M. L. Littman, "Algorithms for Sequential Decision Making," Ph.D. dissertation, Brown University, 1996.

[12] H. Zhang, G. Jiang, K. Yoshihira, H. Chen, and A. Saxena, "Resilient workload manager: taming bursty workload of scaling internet applications," in *ICAC-INDST '09: Proceedings of the 6th international conference industry session on Autonomic computing and communications industry session*. ACM, 2009, pp. 19–28.

[13] E. Kalyvianaki, T. Charalambous, and S. Hand, "Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters," in *ICAC '09: Proceedings of the 6th international conference on Autonomic computing*. ACM, 2009, pp. 117–126.

[14] L. Liu, H. Wang, X. Liu, X. Jin, W. B. He, Q. B. Wang, and Y. Chen, "Greencloud: a new architecture for green data center," in *ICAC-INDST '09: Proceedings of the 6th international conference industry session on Autonomic computing and communications industry session*. ACM, 2009, pp. 29–38.

[15] B. Li, J. Li, J. Huai, T. Wo, Q. Li, and L. Zhong, "Enacloud: An energy-saving application live placement approach for cloud computing environments," in *IEEE International Conference on Cloud Computing*. IEEE Computer Society, 2009, pp. 17–24.

[16] E. Sarhan, A. Ghalwash, and M. Khafagy, "Specification and implementation of dynamic web site benchmark in telecommunication area," in *ICCOMP'08: Proceedings of the 12th WSEAS international conference on Computers*. World Scientific and Engineering Academy and Society (WSEAS), 2008, pp. 863–867.

[17] "Amazon Auto Scaling," http://aws.amazon.com/autoscaling/.

[18] C. J. C. H. Watkins and P. Dayan, "Technical note: Q-learning," *Machine Learning*, vol. V8, no. 3, pp. 279–292, May 1992. [Online]. Available: http://dx.doi.org/10.1023/A:1022676722315

[19] P. Bodik, R. Griffith, C. Sutton, A. Fox, M. I. Jordan, and D. A. Patterson, "Automatic exploration of datacenter performance regimes," in *ACDC '09: Proceedings of the 1st workshop on Automated control for datacenters and clouds*. ACM, 2009, pp. 1–6.