Towards Faster Response Time Models for Vertical Elasticity

Ewnetu Bayuh Lakew, Cristian Klein, Francisco Hernandez-Rodriguez and Erik Elmroth

Department of Computing Science
Umeå University, Sweden
{*ewnetu, cristian.klein, francisco, elmroth*}*@cs.umu.se*

***Abstract*—Resource provisioning in cloud computing is typically coarse-grained. For example, entire CPU cores may be allocated for periods of up to an hour. The Resource-as-a-Service cloud concept has been introduced to improve the efficiency of resource utilization in clouds. In this concept, resources are allocated in terms of CPU core fractions, with granularities of seconds. Such infrastructures could be created using existing technologies such as lightweight virtualization using LXC or by exploiting the Xen hypervisor's capacity for vertical elasticity. However, performance models for determining how much capacity to allocate to each application are currently lacking. To address this deficit, we evaluate two performance models for predicting mean response times: the previously proposed queue length model and the novel inverse model. The models are evaluated using 3 applications under both open and closed system models. The inverse model reacted rapidly and remained stable even with targets as low as 0.5 seconds.**

I. Introduction

The widespread adoption of Infrastructure as a Service (IaaS) cloud computing has partly been driven by one of its defining features: elasticity. Elasticity is the ability to automatically provision and release computing resources, which are usually packaged as Virtual Machines (VMs), rapidly and on-demand. There are two types of elasticity – horizontal and vertical. Horizontal elasticity involves adding or removing VMs to or from an application, e.g. based on its number of end-users. This requires relatively support from the application, e.g. to clone and synchronize states among VMs, but requires no extra support from the hypervisor. It has therefore been widely adopted in public clouds.

Vertical elasticity involves adding or removing resources such as CPU cores and memory to or from a VM. It requires little support from the application, which in essence only needs to be multi-threaded; the elasticity is primarily supported by the hypervisor and the guest operating system's kernel. Horizontal elasticity is generally course-grained. For example, a whole core may be exclusively allocated to a VM for a comparatively long period such as an hour. Conversely, vertical elasticity is fine-grained: individual fractions of a core may be allocated to a VM for as little as a few seconds. Vertical elasticity could thus be a key enabling technology for resource-as-a-service clouds [6], infrastructures that lease resources at CPU-cycle and second granularities. Such infrastructures would benefit cloud users by allowing them to pay only for the resources they actually use, and cloud providers by allowing them to use their resources more efficiently and serve more users. In fact, the technological cornerstone of the resource-as-a-service concept has arguably already been laid by the development of lightweight virtualization frameworks such as LXC [7] and commercial offerings such as dotCloud.

However, the practical exploitation of vertical elasticity will require the creation of methods for determining how much capacity (e.g. how many cores or core fractions) should be allocated to a given application. For example, it is well-known that users are sensitive to response time [15] and that response times of 4 seconds or more may cause them to abandon an application. Much research has been done on resource allocation in horizontally elastic systems. However, the resulting solutions are not directly applicable in the vertical case due to their coarse-grained approach. Other works have examined the vertical case but suffer from various deficiencies; among other things, they may require long training periods of up to 20 minutes [14], [16] (see Section II). We argue that given the accelerating pace of deployment for new versions of cloud applications [18], significantly faster performance models will be required to truly enable the envisioned resource-as-a-service cloud. These models should require only minimal training or knowledge about the hosted applications while simultaneously reacting as quickly as possible to environmental changes such as sudden increases in user numbers.

In this paper we present an empirical study of two mean response time performance models (Section III). The *queue length based* model was proposed in a previous publication and works by measuring the queue length and then applying Little's law. Conversely, our novel *inverted response time* model works by inverting the response time of the M/M/1 queue. We validate both performance models in open- and closed-system models for target response times of as low as 0.5 seconds (Section IV). We show using three popular prototype cloud applications that our inverse response time model outperforms the queue length based approach: it maintains response times around the target value in the absence of environmental changes and reacts within 40 seconds (or 8 control intervals) when changes do occur

while it requires less information from the application. We present our initial findings here in the hope of receiving early feedback from the research community that will facilitate the development of still faster performance models.

II. Related Work

Guitart et al. have extensively surveyed technologies for Internet application performance management [12]. According to their taxonomy, the models examined in this work would be best classified as dynamic resource management technologies on virtualized platforms that use combined approaches in decision making. We gather run-time response time and queue length measurements to fit a queuing model and use a control theoretic approach to filter potential noise and modeling errors. The cited authors noted that comparatively few existing technologies in this area use combined approaches even though such methods arguably show the greatest promise. Our study aims to fill this gap.

We also note two further shortcomings of existing methods for Internet application performance management that were cited in the above survey. First, many have only been validated using simulations or a single application. Second, some proposed methods base their decisions on CPU usage, which is not a reliable measure of spare capacity in vertically elastic systems due to hypervisor preemption of virtual machines, or steal time [9]. In contrast our model bases its decisions on response times and queue lengths, both of which can be measured reliably.

Some more recent approaches to the problem considered herein require up to 20 minutes of training [14] or off-line profiling [16]. As businesses embrace the lean movement, several versions of a given application may be deployed on a daily basis [18], which makes such approaches cumbersome. The development of a method that requires no training and reacts rapidly to changes was thus one of the key objectives of our work. The authors in [10] presented an application-agnostic model-driven autoscaler that tries to ensure performance to be below a certain threshold. However, the approach may lead to resource over-provisioning.

III. Response Time Performance Models

This section begins with a description of our constraints and assumptions. We then present two models for predicting the capacity requirements of a cloud application based on its past behavior and a target response time.

A. Assumptions

Performance models need to satisfy several constraints. First, due to the heterogeneity of hosted applications, the performance models need to be as generic as possible and should require no knowledge of application internals. Second, they should predict average behavior and ignore sporadic noise observed in the past. Such noise may be caused, for example, by variation in data retrieval: some data may be cached in memory, while other information may have to be fetched from disk. Third, these performance models should quickly adjust to variations in workload and capacity requirements. For example, an increase in the number of users or a change in request distribution may necessitate model parameter refitting.

Ideally, when given an input Key Performance Indicator (KPI) value, a performance model should return the exact capacity that must be allocated to an application to achieve that KPI value. Since this is difficult to achieve in practice, a performance model should at least *drive* capacity allocations towards the correct value. That is to say, the model parameters used to estimate capacity requirements should be updated periodically so that the application eventually achieves the desired performance.

The load of a cloud application can be of three types: open, closed or partly-open [19]. In an open-system model, which is typically described using a Poisson process, requests are issued with an exponentially-random inter-arrival time characterized by a rate parameter, without waiting for requests to actually complete. Conversely, in a closed-system model, a number of users access the application, each executing the following loop: issue a request, wait for the request to complete, "think" for a random time interval, repeat. The resulting average request inter-arrival time is the sum of the application's average think- and response times; it therefore depends on the application's performance. A partly-open system model is an intermediate between the open and closed models: users arrive according to a Poisson process and leave after some time, but behave in a closed fashion while in the system. As with the closed model, the inter-arrival time depends on the performance of the evaluated system.

When developing our performance models, we start by assuming an open system. The response time in such a scenario increases more rapidly as the system approaches saturation [19]. Therefore, vertically elastic capacity allocation must be done very carefully. Section IV describes evaluations of the proposed performance models in closed system models to assess their suitability in cases where our assumption of openness does not hold.

B. Response Time Models

End-users of interactive applications are sensitive to response times; several studies have shown that increased response times reduce revenue [15]. It is therefore desirable to maintain a given target response time for each application. Unfortunately, it is difficult to model response times due to their non-linear relationship with capacity. We present two different response time models: the **queue length** model, which has previously only been tested using simulations [8], and our novel **inverse** model.

1) Queue Length Model: Starting from Little's Law, the relationship between the average response time R and

capacity c for an application can be represented as:

$$q = \lambda \cdot R, \tag{1}$$

where q is the average queue length, i.e. the number of requests that have entered the application but have yet to exit, and λ is the arrival rate. Besides, the mean response time given by the M/M/1 queuing model is:

$$R = \frac{1}{\mu - \lambda}, \tag{2}$$

where μ is the application's average service rate. The relationship between capacity and response time can be modeled as $\mu = c/\alpha$, where α is a model parameter. By replacing the λ term in Eq. (2) with the expression for λ from Eq. (1) and rearranging, one obtains the following expression for the mean response time:

$$R = \alpha(q+1)/c, \tag{3}$$

where α is a model parameter and q is the number of requests waiting to be serviced. The parameter α can be estimated online from past measurements of the average response time, average queue length and capacity, thus compensating dynamically for small non-linearities in the real system. However, to reduce the impact of measurement noise, we decided to use a Recursive Least Square (RLS) filter [13]. In essence, such a filter uses past estimates of α and the current ratio $(Rc)/(q+1)$ to output a new value that minimizes the least-squares error. A *forgetting factor* is used to trade the influence of old values for up-to-date measurements. The model has previously only been validated using simulations [8] and has yet to be tested in a real environment.

2) Inverse model: We model the inverse relationship between an application's mean response time R and the capacity allocated to it as:

$$R = \beta/c, \tag{4}$$

where β is a model parameter. As in the queue length model, the parameter β can be estimated using past measurements of capacity and average response time using Eq. (4). As before, to reduce the influence of measurement noise, we use an RLS filter. Note that the inverse model needs less information from the application than the queue length.

In our experiments, we recompute the capacity allocated to the application periodically, with a **control interval** of 5 seconds. This is short enough to make the system reactive and long enough to observe the effects of the new capacity allocation on the application's performance [17] and the overhead of reallocation is negligible. Besides, we use a forgetting factor of 0.2, which was determined experimentally, for both models.

IV. Evaluation

In this section, we evaluate our contribution. First, we describe the experimental setup. Next, we present time series and cumulative analyses of the performance models under different configurations.

A. Setup

Experiments were conducted on a single Physical Machine (PM) equipped with a total of 32 cores[1] and 56 GB of memory. To emulate a typical cloud environment and easily enable vertical elasticity, we used the Xen hypervisor [5]. Each tested application was deployed with all of its components such as web servers and database servers inside its own VM as is commonly done in practice [21], e.g. by using a LAMP stack [1]. Since we were primarily interested in evaluating CPU allocation strategies, we configured each VM with 6 GB of memory, enough to avoid disk activity.

To test our contribution's applicability to a wide range of application types, we performed experiments using three applications: RUBiS [3], RUBBoS [4] and Olio [2]. These applications are widely-used cloud benchmarks (see [11], [20], [23], [22]) and represent an eBay-like e-commerce application, a Slashdot-like bulletin board and an Amazon-like book store, respectively.

To emulate the users accessing the applications, we used our custom `httpmon` workload generator[2], which supports both open- and closed-system model client behavior. For open clients, we changed the arrival rate during the course of the experiments as required to stress-test the system. For the closed case, the think-time of each client was fixed at 1 second and the number of users was varied. The change in arrival rates or number of users was made instantly. This made it possible to meaningfully compare the system's behavior under the two client models. As the application's response time decreases, the throughput of closed clients approaches the value seen for open clients.

Metrics: The **response time** of a request is defined as the time that passes between sending the first byte of the request and receiving the last byte of the reply. We are mostly interested in the mean response time over 20 second intervals (4 control intervals), which are long enough to filter out measurement noise but short enough to reveal an application's transient behavior.

B. Time series analysis

Experiments were performed using variable loads to characterize the performance models' responses to sudden workload spikes under the open- and closed-system models. The target response times used in the experiments ranged from relatively high to quite low in order to assess the performance models' behavior in each case.

[1]Two AMD Opteron™ 6272 processors, 2100 MHz, 16 cores each, no hyper-threading.

[2]https://github.com/cloud-control/httpmon

The plots in this section are structured as follows. Each figure shows the results of a single experiment. The bottom x-axis represents the time elapsed since the start of the experiment. The time is divided in 5 equal intervals, each featuring a different arrival rate (for the open system model) or number of users (for the closed system model). The arrival rates or user numbers during each such interval are presented on the top x-axis. The upper graph in each figure plots the measured mean response time while the lower graph plots the capacity required over the next 5 second interval as computed by the performance model and allocated to the application.

Figs. 1a to 1d show the performance of the two models when configured with different target response times under open- and closed-system models for the RUBiS application. In general, both performance models are stable with higher target response times under both system models. Moreover, both models converge to the target values quite quickly (see Section IV-D) after detecting a sudden increase or decrease in workload (manifested as a rapid increase or decrease in the response time).

It should be noted that both models correctly detect and adapt to the capacity requirements of both open- and closed-system models. At higher target response times, the open-system model requires more capacity than the closed-system model when using comparable arrival rates and user numbers, respectively. For lower target response times, the capacity requirements for the two system models become almost identical. These situations are properly dealt with by both performance models, as shown in Fig. 1a and Fig. 1c for higher targets, and Fig. 1b and Fig. 1d for lower targets.

We also performed experiments with Olio and RUBBoS. Due to space constraints, we only present time series plots generated using a target response time of 0.5 for these applications. As shown in Figs. 2 and 3, the queue length model does not behave well for lower target values with these applications but the inverse model remains stable.

In general, the inverse model remains relatively stable irrespective of the target value under both system models. Conversely, the queue length model is less stable for lower target values under both system models.

The time series results show that the models behave as intended. In the following sections we analyse their cumulative behaviors such as the aggregate errors over the span of the experiment (Section IV-C) and the total time it takes each model to reach a stable state after a change in load (Section IV-D).

C. Aggregate Analysis

The performance models' aggregate behavior over the course of each experiment was assessed using two control theoretic metrics that measure the total error observed during the system's life span. These metrics are Integral of Squared Error (ISE) and Integral of the Absolute Error (IAE), which are computed as shown below:

$$ISE = \sum (e(t))^2, \tag{5}$$

$$IAE = \sum (e(t))^2, \tag{6}$$

Table I: Errors of the inverse and queue-length models for RUBiS.

Target [seconds]	System Model	Performance Model	ISE	IAE
1.5	open	inverse	80.20	34.80
		queue length	73.61	32.62
	closed	inverse	148.07	70.47
		queue length	131.14	64.71
1.0	open	inverse	55.04	29.82
		queue length	43.36	26.87
	closed	inverse	119.40	67.05
		queue length	103.30	72.98
0.5	open	inverse	6.01	75.23
		queue length	88.38	85.75
	closed	inverse	43.68	57.05
		queue length	110.76	100.69
0.1	open	inverse	2.60	8.91
		queue length	7.11	14.33
	closed	Inverse	2.05	7.05
		queue length	2.21	8.74

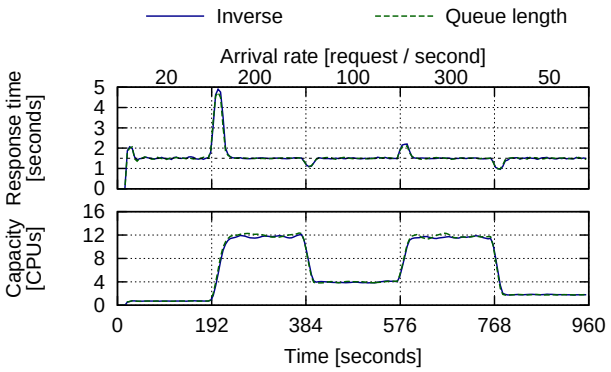
Table II: Errors of the inverse and queue-length models for Olio and RUBBoS with a 0.5s target.

Application	System Model	Performance Model	ISE	IAE
Olio	open	inverse	3.48	17.26
		queue length	4.89	21.34
	closed	inverse	9.04	31.42
		queue length	187.94	119.70
RUBBoS	open	inverse	10.19	14.65
		queue length	9.95	16.61
	closed	inverse	50.78	53.76
		queue length	319.86	160.25

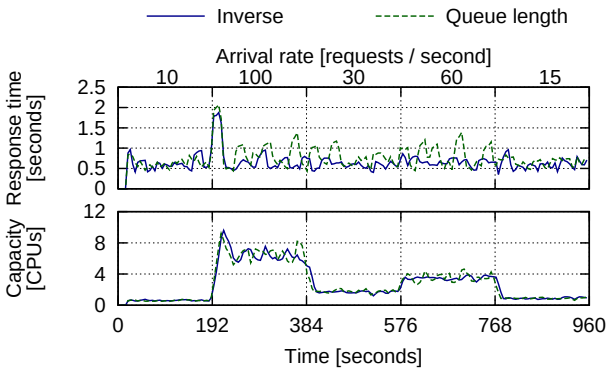
Tables I and II show the aggregate errors of both performance models for the three applications with different target response times and system models. Under the open system model, both the ISE and IAE for the queue length model are smaller than those for the inverse model when the target value is relatively high. The opposite is true for lower target values. This implies that in an open system, the queue length model is slightly preferable for higher target response times while the inverse model is preferable for lower target values. Under the closed system model, the inverse model's ISE and IAE error values are lower than those for the queue length model irrespective of the target response time. The inverse model is thus preferable for closed systems.

D. Adaptation Period

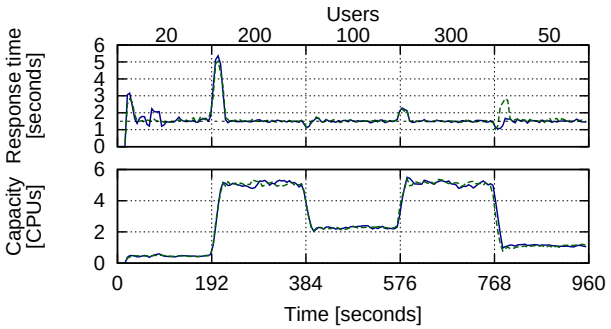
The adaptation period is the amount of time required for each model to converge to the target response time in a worst-case scenario following a change in the state of the system (i.e. in the number of users). The system was considered to have converged if the deviation of the measured response time was within 10% of the target value. Table III shows the adaptation periods of each performance model for different applications. At higher target values, the system always converges within 10 to 30 seconds. With lower targets, the queue length model does not converge according



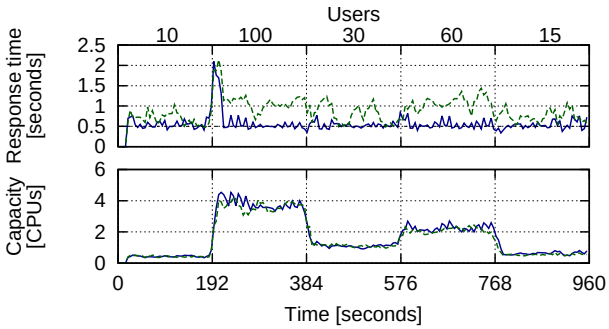
(a) open system model, 1.5s target



(b) open system model, 0.5s target

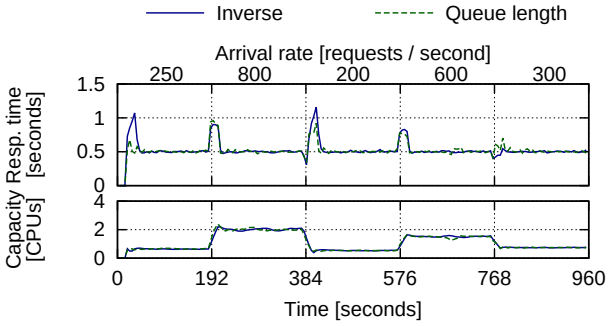


(c) closed system model, 1.5s target

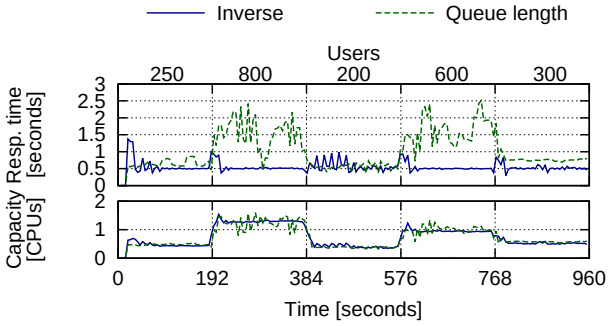


(d) closed system model, 0.5s target

Figure 1: Capacity allocation and response times for RUBiS under open and closed system models with 0.5s and 1.5s target response times.

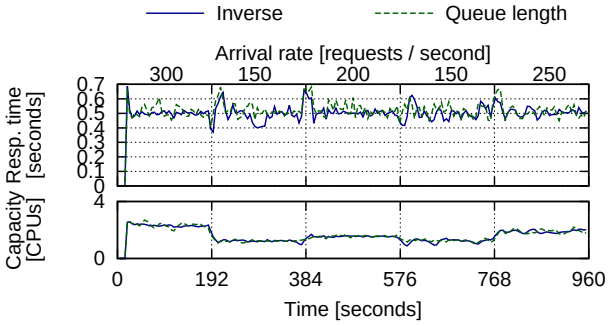


(a) open system model, 0.5s target

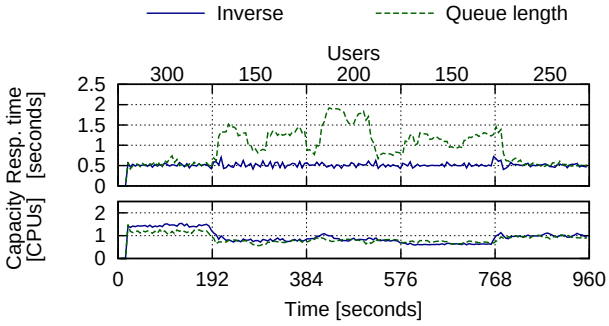


(b) closed system model, 0.5s target

Figure 2: Capacity allocation and response times for RUBBoS under open and closed system models with a 0.5s target response time.



(a) open system model, 0.5s target



(b) closed system model, 0.5s target

Figure 3: Capacity allocation and response times for Olio under open and closed system models with a 0.5s target response time.

to the above definition because it does not maintain response times close enough to the target. The inverse model failed to keep the response time close to the target value under the open system model for RuBiS but did achieve convergence within 40 seconds under the closed system model.

Table III: Adaptation periods for the three applications.

Application	Target [seconds]	Adaptation period [seconds]			
		Open		Closed	
		Inverse	Queue length	Inverse	Queue length
Olio	1.5	20	20	20	20
	0.5	10	10	5	–
RUBBoS	1.5	10	20	10	10
	0.5	20	35	40	–
RUBiS	1.5	25	25	20	25
	1.0	25	20	30	30
	0.5	–	–	10	–

E. Discussion

Our experiments showed that both models perform well with relatively high response time targets under both closed- and open-system models. However, the inverse model performs better for lower response time targets. More specifically, our results yielded the following **key findings**:

1) Both performance models properly predict the capacity required for both open- and closed-system models.
2) Both performance models are more stable when the response time target is relatively high. However, the inverse model is more stable than the queue-length model for lower targets.
3) The inverse model is more stable under closed system model than under open system model for lower targets.
4) Both models reach stability very quickly (i.e. within 40 seconds) after detecting a change in the system.

V. Conclusion

We have presented two generic mean response time performance models that map performance to capacity in order to provide performance guarantees for interactive applications deployed in the cloud. Both models were tested in an extensive set of experiments using different applications with workload mixes that varied over time under both closed- and open-system models. We also varied the target response times of each application to see how this affected the models' behavior. Our results demonstrate that both performance models are stable for higher response time targets. However, our new inverse model is more stable than the queue length model for lower targets. Furthermore, our inverse model converges within 40 seconds, which is substantially less than the 20 minutes required by state-of-the-art alternatives. Our contribution thus paves the way to effective capacity allocation for vertical elasticity, enabling the future Resource as a Service (RaaS) cloud.

Future work in this area will focus on improving the stability and reaction time of our inverse model and extending it to handle the response time tail.

Acknowledgement

This work was partially supported by the Swedish Research Council (VR) project *Cloud Control* and the Swedish Government's strategic effort *eSSENCE*.

References

[1] Tutorial: Installing a LAMP web server, 2013. available online: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/install-LAMP.html.

[2] Olio, 2014. Available online: http://incubator.apache.org/projects/olio.html.

[3] Rice university bidding system, 2014. Available online: http://rubis.ow2.org.

[4] Rubbos, 2014. Available : http://jmob.ow2.org/rubbos.html.

[5] P. Barham et al. Xen and the art of virtualization. In *SOSP*. ACM, 2003.

[6] Orna Agmon Ben-Yehuda et al. The rise of RaaS: The resource-as-a-service cloud. *Commun. ACM*, 57(7), 2014.

[7] S. Bhattiprolu et al. Virtual servers and checkpoint/restart in mainstream linux. *SIGOPS OS Rev.*, 42(5):104–113, 2008.

[8] A. Chandra et al. Dynamic resource allocation for shared data centers using online measurements. In *IWQoS*, pages 381–398. Springer, 2003.

[9] C. Ehrhardt. Cpu time accounting. http://www.ibm.com/developerworks/linux/linux390/perf/tuning_cputimes.html. Last accessed: Aug. 2013.

[10] Anshul Gandhi et al. Adaptive, model-driven autoscaling for cloud applications. In *ICAC*, pages 57–64. USENIX, 2014.

[11] Z. Gong et al. PRESS: Predictive elastic resource scaling for cloud systems. In *CNSM*. IEEE, 2010.

[12] J. Guitart and others. A survey on performance management for internet applications. *Concurr. Comput.: Pract. Exper.*, 22(1):68–106, 2010.

[13] W. Liu et al. *Kernel Adaptive Filtering: A Comprehensive Introduction*. Wiley Publishing, 1st edition, 2010.

[14] L. Lu et al. Application-driven dynamic vertical scaling of virtual machines in resource pools. In *NOMS*, 2014.

[15] F. Fui-Hoon Nah. A study on tolerable waiting time: how long are web users willing to wait? *Behaviour and Information Technology*, 23(3), 2004.

[16] H. Nguyen et al. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *ICAC*, 2013.

[17] P. Padala et al. Automated control of multiple virtualized resources. In *EuroSys*. ACM, 2009.

[18] Eric Ries. *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*. Crown Business, 2011.

[19] B. Schroeder et al. Open versus closed: A cautionary tale. In *NSDI*, 2006.

[20] Z. Shen, , et al. CloudScale: elastic resource scaling for multi-tenant cloud systems. In *SoCC*. ACM, 2011.

[21] K. Sripanidkulchai et al. Are clouds ready for large distributed applications? *SIGOPS Oper. Syst. Rev.*, 44(2), 2010.

[22] C. Stewart et al. Exploiting nonstationarity for performance prediction. In *EuroSys*. ACM, 2007.

[23] N. Vasić et al. DejaVu: accelerating resource allocation in virtualized environments. In *ASPLOS*. ACM, 2012.