Tail Response Time Modeling and Control for Interactive Cloud Services

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*—**

Services hosted in the cloud have become indispensable for many business and personal processes, making performance of these services a key issue. However, today, cloud infrastructure providers either do not offer any performance guarantees or prefer coarse-grained and static VM provisioning with fixed sizes resulting inefficient resource utilization and SLO violations. In order to increase resource utilization and guarantee strict performance requirements of services, fine-grained resource allocation techniques are required. In essence, resources can be allocated in terms of CPU core fractions, with granularities of seconds according to the service needs. Such infrastructures may be created using existing technologies such as lightweight virtualization using Linux Containers (LXC) or by exploiting the Xen hypervisor's capacity for vertical elasticity.

Fine-grained resource provisioning entails determining the amount of resources required to meet each hosted service's performance. However, current performance models for determining how much capacity to allocate to each service cannot be readily used, due to their course-grained assumptions. To address this deficit, we proposed two performance models for predicting capacity that guarantees tail response times: the queue length tail model and the inverse tail model. Differently from most of the current approaches, where the performance guarantees are characterized only in terms of bounds on average response time, we consider Service Level Objectives that are specified using tail values such as 95% or 99%, which are considered better correlated to user experience. The models were evaluated using 3 services under both real and synthetic workloads generated based on open- and closed-system models. The inverse percentile model shows to perform rapid adjustment actions and provides stable performance even under unpredictable and spiky workloads.

I. Introduction

Cloud resources are widely used by major IT companies and startups to remain competitive. They provide a pool of computing resources that can be dispatched to services on-demand based on pay-per-use pricing schemes [23]. The resources are virtualized and can be rapidly provisioned and released depending on applications' demands without human intervention.

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 provisioned in the form of 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 a service, e.g. based on dynamic resource needs. This requires 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 on the other hand involves adding or removing resources such as CPU cores and memory to or from an individual VM. It requires little support from the service, which in essence only needs to be multi-threaded; the elasticity is primarily supported by the hypervisor and the guest operating system's kernel.

Most current commercial cloud infrastructure providers either do not offer any performance guarantees and resources are provisioned based on horizontal scaling, which is coarse-grained with static and fixed VM size configurations. This leads to inefficient resources utilization as well as SLO violations due to long instantiation latency [25]. In order to increase resource utilization and guarantee strict performance requirements of services, fine-grained resource allocation techniques based on vertical elasticity are required. In essence, resources can be allocated in terms of CPU core fractions, with granularities of seconds according to the service needs.

Vertical elasticity could thus be a key enabling technology for resource-as-a-service clouds [11], infrastructures that lease resources at CPU-cycle and second granularities. Such infrastructures would benefit cloud users by allowing them to use resources that meet the performance needs of their services and 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 [12] or by exploiting the Xen hypervisor's capacity for vertical elasticity. Commercial offerings such as dotCloud[1] and ProfitBricks[2] have already started supporting vertical elasticity

[1] https://www.dotcloud.com/
[2] http://www.profitbricks.com

which is considered one of the important features of the second generation IaaS (IaaS 2.0) [3].

Vertical elasticity entails determining the amount of resources required to meet each hosted service's performance targets, e.g., possibly included in a formal Service Level Agreement (SLA). Nonetheless, maintaining services' performances targets is a challenging task that requires complex decisions within short time periods. This is because of the intrinsically dynamic and unpredictable nature of the services, workloads and operating environments as well as non-trivial relationship between the performance metrics (e.g., throughput or response time) and amount of resources. Moreover, 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 service to maintain, for example, response time within a bound. It has been shown [24], [1], [2] that a web page response times exceeding four seconds leads to a loss of at at least 25% of the visitors, and for e-commerce, every 100 ms extended response time reduces sales 1%.

Much research has been done on resource allocation in horizontally elastic systems [15]. However, the resulting solutions are not directly applicable in vertical case due to their coarse-grained approach and longer time scale [22], [25]. Other works have examined vertical case but suffer from various deficiencies; among other things, they may require long training periods of up to 20 minutes [22], [25] (see Section II). Given the accelerating pace of deployment for new versions of services [27], we previously proposed and evaluated [20] a significantly faster mean response time models that can be used to truly enable the envisioned resource-as-a-service cloud. These models require only minimal training or knowledge about the hosted services while simultaneously reacting as quickly as possible to environmental changes such as sudden increases in user numbers or service upgrades. Nonetheless, the above vertical elasticity approaches have a major limitation in the fact that they are only designed for performances that are expressed in terms of mean values instead of tail values which are better correlated with user experience [18]. For example, they fail to maintain 95-th or 99-th percentile of the response times under a certain target, which is actually a much more important performance target than the average response time. This is because tail response time generates very noisy signals which make it very challenging to keep it constant around a target value. To the best of our knowledge, our current contribution is the first attempt to develop a stable and faster model for controlling tail response time.

In this paper we present two novel performance models, referred to as the *queue length tail* and *inverse tail* models, for controlling tail response time (Section IV). The *queue length tail* model extends the mean response time model proposed in [13] and works by measuring the queue length and then applying Little's law and M/M/1 to find the relationship between response time and capacity. The *inverse tail* is based on our own mean response time model [20] and works using the inverse relationship between response time and capacity. We validated both performance models using Wikipedia and FIFA traces and synthetic traces generated based on open- and closed-system models for target response times (Section V). The performance of the models is evaluated using three popular prototype cloud services. The results show that the inverse tail response time model outperforms the queue length tail model: it maintains tail response times around the target value under environmental changes (both workloads and service internal behavior changes) and reacts within 40 seconds (or 8 control intervals) even for major changes in workloads while it requires less information from the service compared to the queue-length tail model.

II. Related Work

Guitart et al. have extensively surveyed technologies for Internet services performance management [19]. According to their taxonomy, the models examined in our current 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. Guitart et.a.l noted that comparatively few existing technologies in this area use combined approaches even though such methods arguably show the great potential. 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 [14]. 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 [22] or off-line profiling [25]. As businesses embrace the lean movement, several versions of a given application may be deployed on a daily basis [27], 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 [16] presents 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.

The authors in [20], [13] propose a significantly faster mean response time models with online parameter estimation techniques. These models 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 or application upgrades. Nonetheless, their major limitation is that they are only designed for performances that are expressed in terms of mean values instead of percentile (tail) values. For example, they fail to maintain 95 % or 99 % of the response times under a certain target.

III. Overview of the Autonomic Capacity Controller

We consider a cloud infrastructure that hosts interactive services, each with unpredictable and variable workload dynamics. Each service has a performance target (or even an SLA) associated with them that stipulates a target value for a Key Performance Indicator (KPI), expressed as *response time* along with the *percentile* value such as 95% or 99% to be controlled.

The goal is then to continuously adjust capacity allocations to services automatically without human intervention, so as to drive performance towards the given targets. Specifically, the desired autonomic capacity controller should be capable of allocating just the right amount of capacity for each service in order to meet their respective performance target, avoiding both capacity under- and over-provisioning.

Fig. 1 shows the architecture of the proposed autonomic capacity controller implemented on a single Physical Machine (PM) to meet tail response time values for hosted services. It loosely follows a Monitor, Analysis, Planning and Execution (MAPE) loop. Monitoring is responsible for retrieving information from the hosted services about their observed tail response time R_t, mean response time R_a and queue length q which are input for the controllers. During analysis, the capacity required by a service to meet its performance target, expressed in tail response time value, is computed using the performance models in two steps.

In the first steps, the response time controller estimates the mean response time R_a', which in turn is used to estimate capacity R_t during the second step, from the target tail response time R_t' as well as observed tail and mean response times. The purpose of this step is to stabilize the system by eliminating the noisy behavior of the tail response time values. In the second step, the capacity required to meet the target is estimated using the selected mean response time model (i.e., inverse or queuing model). Past measurements are used to fit model parameters, which are similar to the Knowledge component of autonomic controllers. Finally, during the planning and execution phase, the hypervisor is configured to enforce the computed capacities. The high-level function of each the components depicted on Fig. 1 is described as follows:

- *Response time controller.* Dynamically determines the relationship between tail response time and average response time. It acts as stabilizer by eliminating the noise from the tail response time values. The output of the controller is average response time which is used as input to the capacity controller. Specifically, given the tail response time, it predicts the value of the average response time to be used during capacity allocation in the second step (Section IV).
- *Capacity controller.* A self-adaptive controller that dynamically adjusts the capacity that should be allocated to each service depending on its workload dynamics (Section IV). It allocates the right amount of capacity for each service in order to meet their respective performance targets.
- *Model Estimator.* Estimates model parameters which captures the complex relationship between application performance and capacity. The model parameters are estimated online from past measurements of the tail and average response times, 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 Recursive Least Squares (RLS) filter [21]. In essence, such a filter uses past estimates of the parameters together with performance measurements during the current control interval to output new parameter values, which is used for the next control interval. The RLS filter minimizes the least-squares error (Section IV).
- *Monitoring.* Monitoring gathers performance information such as queue lengths, average response times and tail (percentile) response times for each service. The performance information is sent to the *model estimator* and *response time controller* as an input for parameter estimations and further control decisions every control interval.

To summarize the process, the autonomic capacity controller periodically polls the measured performance information from each service. We refer to this period as the control interval. The performance values for each service are then used as feedback in the decision making process for the next control interval.

IV. Response Time Performance Models

This section begins with a description of our constraints and assumptions. We then present the tail response time model along with the two average response time models, which are used in the second stage for predicting the capacity requirements of a cloud application based on its past behavior and estimated average 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

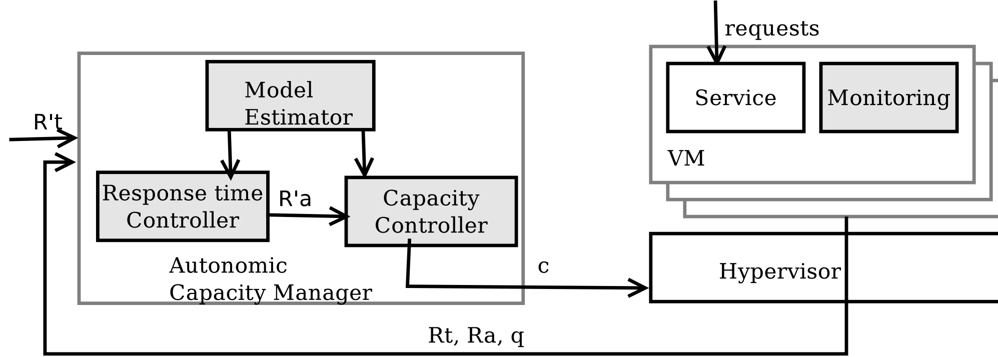


Figure 1: Architecture of the percentile-based autonomic capacity controller on a single physical machine. Modules in gray are part of our contribution.

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 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 due to changes in workloads and unpredictability of applications' internal behavior, 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 [28]. 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 [28]. Therefore, vertically elastic capacity allocation must be done very carefully. Section V 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. Tail of Response Time Models

End-users of interactive applications are sensitive to response times; several studies have shown that increased response times reduce revenue [24]. 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. Our goal is to model tail of response times. We select tail response time since it is better correlated to end-user experience than averages, and tails are not handled well by current performance modeling techniques.

In our previous work [20], it has been shown that two mean response time models, *queue length* and *inverse*, provide a better estimation of capacity. However, using these two models as-is to estimate capacity for tail response time failed to deliver the desired result due the noisy nature of tail values. As a result, we developed a novel approach that dynamically and continuously adjusts the relationship between tail and mean response times. Using the relationship, the system first predicts the target mean response time from the target tail response time, i.e., the mean response time that the application should observe to achieve the tail response time stipulated in the SLA. In effect, what this step does is to eliminate the noisy signals of the tail values. Second, the capacity requirement is predict based on either of the two mean response time models (See Sections IV-C1 and IV-C2) using the mean response time value computed during the first stage.

The relationship between the tail of response time R_t and the average response time R_a is modeled as:

$$R_t = \gamma R_a, \tag{1}$$

where γ is a model parameter. Given R_t, R_a can be computed, which is then used to compute the capacity using either the

queue length model (i.e., Eq. (4)) or inverse model (i.e., Eq. (5)).

The parameter γ can be estimated online from past measurements of the tail response time, average response time, thus compensating dynamically for small non-linearities in the real system. However, to reduce the impact of measurement noise, we decided to use a RLS filter [21]. In essence, such a filter uses past estimates of γ and the current ratio $(R_t)/(R_a)$ to output a new value that minimizes the least-squares error. A *forgetting factor* f_{tail} is used to trade the influence of old values for up-to-date measurements.

The following section, provides brief discussion for the two mean response time models which are used during the second stage.

C. Average Response Time Models

We present the two different response time models: the **queue length** model, which was proposed by [13], and our **inverse** model presented in [20].

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:

$$q = \lambda \cdot R, \tag{2}$$

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:

$$R = \frac{1}{\mu - \lambda}, \tag{3}$$

where μ is the application's average service rate. The relationship between capacity and average service rate can be modeled as $\mu = c/\alpha$, where α is a model parameter. By replacing the λ term in Eq. (3) with the expression for λ from Eq. (2) and rearranging, one obtains the following expression for the mean response time:

$$R_a = \alpha(q+1)/c, \tag{4}$$

where α is a model parameter and q is the number of requests waiting to be serviced. As in the tail model, the parameter α is continuously estimated using past measurements of the average response time, average queue length and capacity. In the same way, to reduce the impact of measurement noise, we decided to use a RLS filter with a forgetting factor f_{ql}. Thus, the filter uses past estimates of α and the current ratio $(Rc)/(q+1)$ to output a new value that minimizes the least-squares error.

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

$$R_a = \beta/c, \tag{5}$$

where β is a model parameter. As in the tail and queue length models, the parameter β can be estimated using past measurements of capacity and average response time using Eq. (5). As before, to reduce the influence of measurement noise, we use an RLS filter with a forgetting factor f_{inv}. Note that the inverse model needs less information from the application than the queue length.

In our experiments, we recompute the capacity allocated to services periodically, with a **control interval** of 10 seconds. This is short enough to make the system reactive and long enough to observe the effects of the new capacity allocation on the service's performance [26] and the overhead of reallocation is negligible. Besides, the model parameters were estimated at every control interval. We used the forgetting factors $f_{\text{ql}} = 0.25$, $f_{\text{inv}} = 0.45$ and $f_{\text{tail}} = 0.85$, which were determined experimentally, for queue length, inverse and tail models respectively.

V. 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 PM equipped with a total of 32 cores[3] and 56 GB of memory. To emulate a typical cloud environment and easily enable vertical elasticity, we used the Xen hypervisor [10]. 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 [30], e.g. by using a LAMP stack [4]. 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 [7], RUBBoS [8] and Olio [5]. These applications are widely-used cloud benchmarks (see [17], [29], [32], [31]) and represent an eBay-like e-commerce application, a Slashdot-like bulletin board and an Amazon-like book store, respectively.

B. Workloads

Experiments were performed using different workloads to characterize the performance models' responses to workload changes. We tested the models using real workload extracted from traces and synthetic workload generated based on the open and closed-system models [28]. The real workloads were extracted from the Wikipedia [6] and FIFA [9] traces. These two traces were selected due to their complementary nature. While the Wikipedia workload shows a steady and

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

predictable trend, the FIFA shows a bursty and an unpredictable trend. For each service, we selected a representative day as shown in Figs. 2a and 2b, to create more different trends and peaks, and showing that not all the services may have the same usage patterns.

In addition, synthetic workloads were generated based on the open- and closed-system models that gave us the freedom of evaluating parameters that were not controlled in the real world workloads. For instance, to increase the number of requests by five-folds or ten-folds to understand the behavior of our solution. We also kept the number of requests constant for some time to study the behavior of the models under the steady- and transient-states.

To emulate the users accessing the applications, under the synthetic workload, we used our custom `httpmon` workload generator[4], 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 tail and mean response time over 10 second intervals, for the synthetic workload, and 1 minute intervals (6 control intervals), which are long enough to filter out measurement noise but short enough to reveal an application's transient behavior.

C. Time series analysis

The target response times used in the experiments were varied in order to assess the performance models' behavior under different scenarios. We set the percentile value to either 95% or 99% throughout our experiments since these are commonly used percentile values.

1) Real Workloads: 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 upper graph in each figure plots the measured tail and mean response times (See Eq. (1)) while the lower graph plots the capacity required as computed by the performance model and allocated to the application.

Figs. 3 and 4 show the performance of the two tail response time models under steady and predictable Wikipedia workload for RUBiS and RUBBoS services respectively. The top figures depict the results of the inverse tail model while the bottom figures depict the results of the queue length tail model. Besides, the figures on right side show the results for 95% while the figures on left side show the result for 99%. The results indicate that both models predict the right amount of capacity required to satisfy the demand in spite of the changes in workload as seen from both Figs. 3 and 4.

However, close observation shows that the results in Fig. 3b and Fig. 4b exhibit more oscillation around the target than the corresponding results in Fig. 3a and Fig. 4a respectively. The same is true for Fig. 3d and Fig. 3c as well as Fig. 4d and Fig. 4c. The is due to the fact that maintaining very high tail value such as 99% is more challenging than relatively low value such as 95%. Besides, the results also show that the amount of capacity required to maintain 99% (See Figs. 3b, 3d, 4b and 4d) is marginally higher than the amount required for 95% (See Figs. 3a, 3c, 4a and 4c) for comparable loads. Further observation on Figs. 3 and 4 shows that the measured response time was smoother under the inverse tail model than the queue length tail model indicating its better prediction. The other observation to note is that the shapes of the capacity graphs depicted on Figs. 3 and 4 follow similar trend as the corresponding workload graph depicted on Fig. 2a for both RUBiS and RUBBoS services.

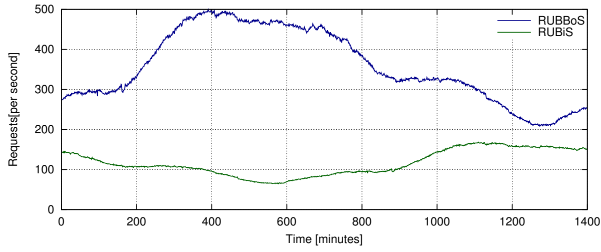
Figs. 5 and 6 show the performance of the two tail response time models under more unpredictable and irregular FIFA workload for RUBiS and RUBBoS applications respectively. In general, both performance models behave as intended even under unpredictable workloads. For example, the high spikes present in the workloads between 900–1300 minutes for RUBiS (Fig. 5) and between 200–700 minutes for RUBBoS (Fig. 6) were handled by both models fairly well.

The results indicate that the performance observed under queue length tail model showed more irregularity and oscillation around the target than the one observed under inverse tail model. In general, under unpredictable workloads, as can be seen from Figs. 5 and 6, the inverse tail model was more stable and able to predict capacity better than the queue length tail model.

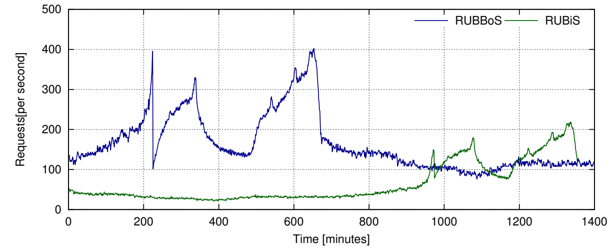
We also performed similar experiments for Olio service. We opted not to present the results here due to their similarity to RUBBoS' and RUBiS'. The inference made from the results is that the inverse tail model shows better stability and prediction both under the predictable Wikipedia as well as irregular and unpredictable FIFA workloads.

2) Synthetic Workloads: 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 into 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

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

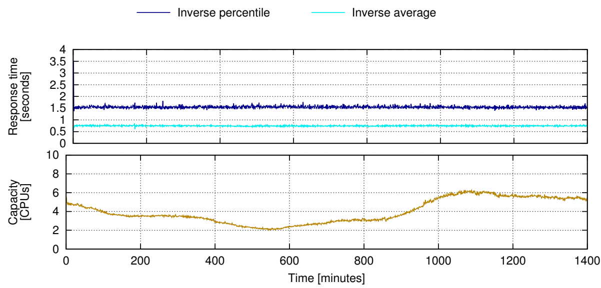


(a) Predictable, Wikipedia-based

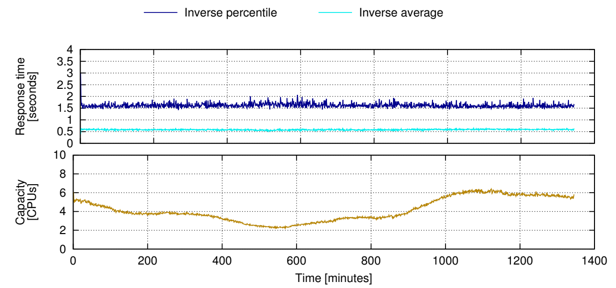


(b) Unpredictable,FIFA-based

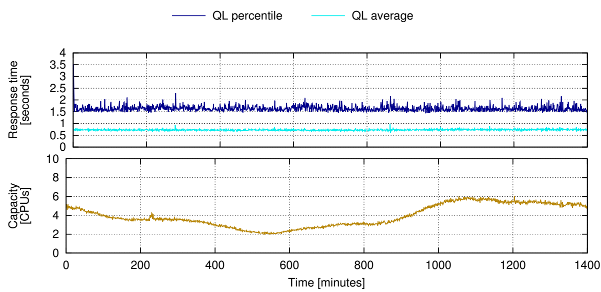
Figure 2: Workloads from real world traces.



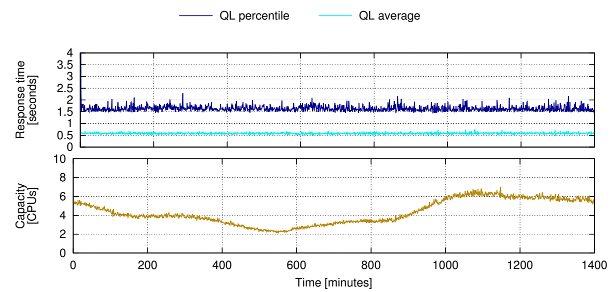
(a) Inverse model with 95% target



(b) Inverse model with 99% target

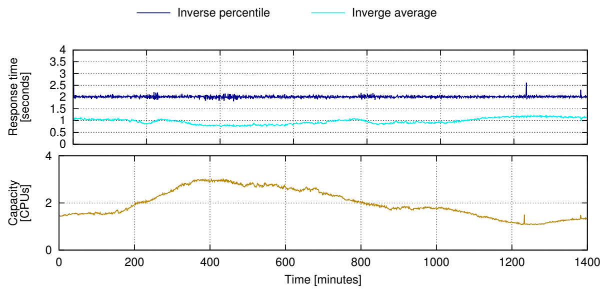


(c) QL model with 95% target

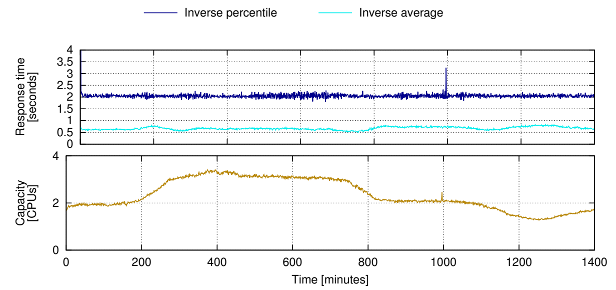


(d) QL model with 99% target

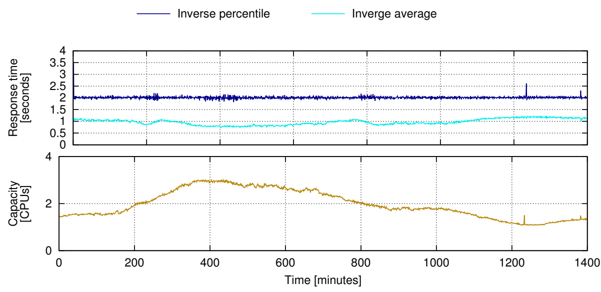
Figure 3: Response times of the two tail models under Wikipedia workload with target response time value of 1.5 sec for RUBiS service.



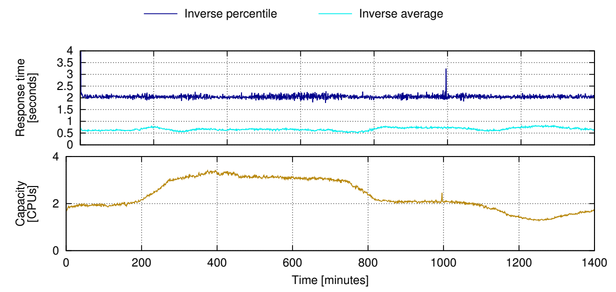
(a) Inverse model with 95% target



(b) Inverse model with 99% target



(c) QL model with 95% target



(d) QL model with 99% target

Figure 4: response times of the two tail models Under Wikipedia workload with target response time value of 1.0 sec for RUBBoS service.

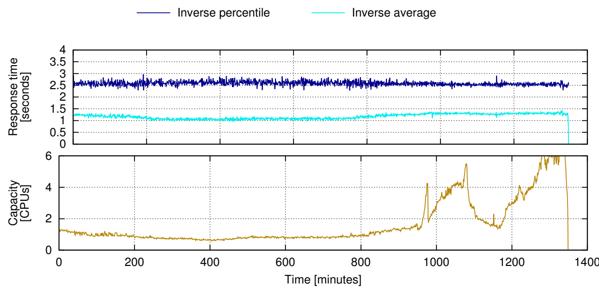
during each such interval are presented on the top x-axis. The upper graph in each figure plots the measured tail and mean response time while the lower graph plots the capacity required over the next 10 second interval as computed by the performance model and allocated to the application.

Figs. 7 to 9 show the performance of the two models when configured with different target response times either for 95% or 99% under open- and closed-system models for the RUBiS service. In general, both performance models were stable with higher target response times under both system models (see Fig. 7). However, slight oscillation was observed for both performance models especially when the percentile was set to 99% under closed-system model while the observed performance was very smooth under open-system model. Moreover, both performance models converge to the target values quite quickly (see **??**) after detecting a sudden increase or decrease in workload (manifested as a rapid increase or decrease in the response time) as depicted in the figures at the beginning of each interval.

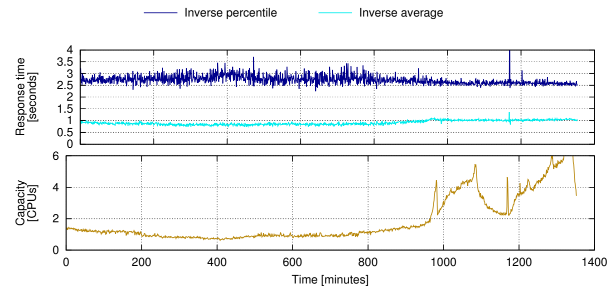
It should be noted that both performance models correctly detect and adapt to the capacity requirements of both open- and closed-system models. At higher target response times, the open-system model required more capacity than the closed-system model when using comparable arrival rates and numbers of users, respectively. As the target response times is lowered, the difference in the capacity required between the two system models decreases. These situations were properly dealt with by both performance models, as shown in Figs. 7a and 7b and Figs. 7c and 7d for higher targets, and Figs. 9a and 9b and Figs. 9c and 9d for lower targets.

We also performed experiments with d RUBBoS. Due to the similarity of the results to RUBiS application, we only present time series plots generated using a target response time of 1.5 and 95% for these applications. As shown in Fig. 10, the results show that the performance models behave as intended.

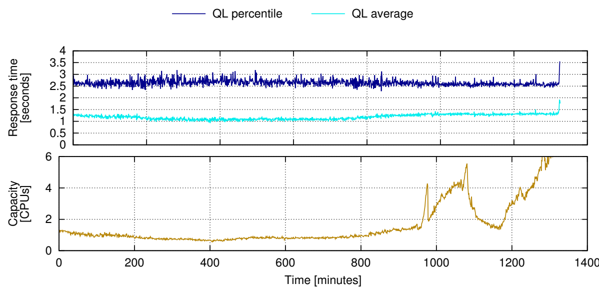
In general, the results show that the inverse model remains relatively stable irrespective of the target value under both system models. Moreover, the inverse model is more stable



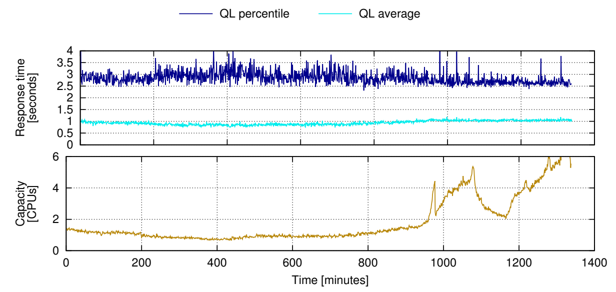
(a) Inverse model with 95% target



(b) Inverse model with 99% target



(c) RUBiS: QL model with 95% target



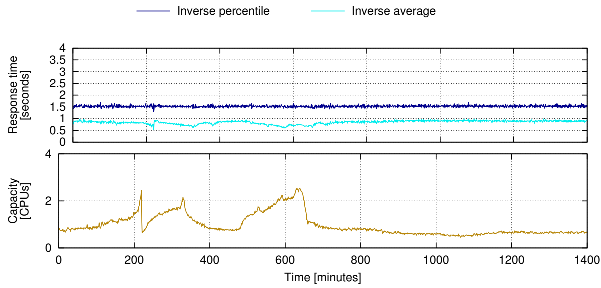
(d) QL model with 99% target

Figure 5: Response times of the two tail models Under FIFA workload with target response time value of 2.5 sec for RUBiS service.

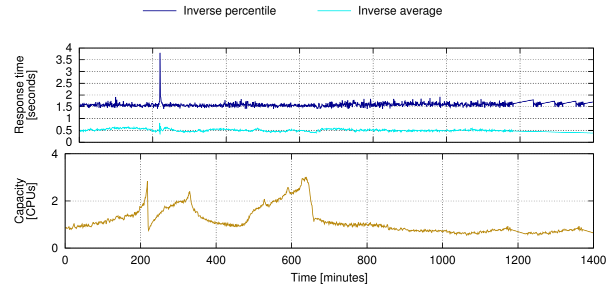
under closed system model than open system model for lower target values while the reverse is true for higher target values. Conversely, the queue length model is less stable for lower target values under both system models. Besides, the results show that controlling 99% is more challenging than 95% and 99% demands slightly higher capacity than 95% for comparable arrival rate or number of users.

3) Changes inside the application: The results presented in the previous sections show the behavior of the models under external changes (i.e., dynamic changes in workloads). In this section, we present the effect of internal changes in the application on the performance models. To demonstrate this, we implemented a mechanism that offers product recommendations with a configured probability, that we call the **recommendation ratio**. A recommendation ratio of 0 means that all requests are served without recommendations and the application has lower capacity requirements, while a value of 1 means that all requests are served with recommendations and the application has higher capacity requirements.

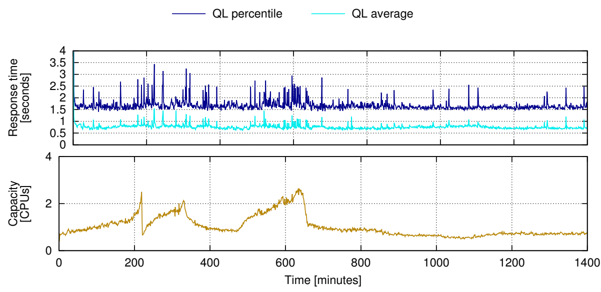
Fig. 11 shows the results for the two performance models when the probability values were varied across five different intervals for RUBiS application with 95% configuration. The arrival rates for open-system model or the number of users for closed-system model was kept the same throughout the experiment. In each figure the upper graph plots the measured response times, the middle graph plots the probability value and the lower graph plots the capacity required as computed by the performance model and allocated to the application. The results reveal that the two models behave fairly well even under changes inside the application. The capacity predicted were in-line with changes in the recommendation ratio. For example, highest recommendation ratio requires the most capacity to satisfy the demand (see interval 2 of Fig. 11a or Fig. 11b) while lowest recommendation ratio implies the least capacity demand (see interval 5 of Fig. 11a or Fig. 11b). And this is perfectly reflected across the five intervals. The performance results for both performance models under open-system model were relatively stable and smooth compared to their corresponding results under closed-system model. However, the queue length tail



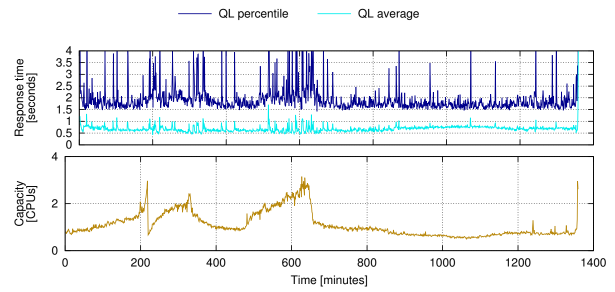
(a) Inverse model with 95% target



(b) Inverse model with 99% target



(c) QL model with 95% target



(d) QL model with 99% target

Figure 6: Response times of the two tail models Under FIFA workload with target response time value of 1.5 sec for RUBBoS service.

model under closed-system model was the most unstable and oscillatory. Overall, the inverse tail model was relatively stable under both system models. In general, the result indicates that the models can be used in wide range of scenarios without much modification.

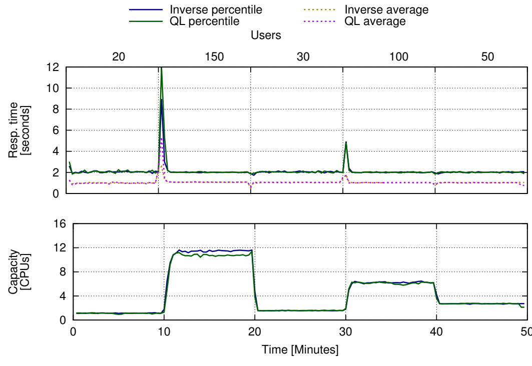
D. Aggregate Analysis

In this section we analyse cumulative behaviors using statical (Section V-D1) and control theoretic (Section V-D2) techniques.

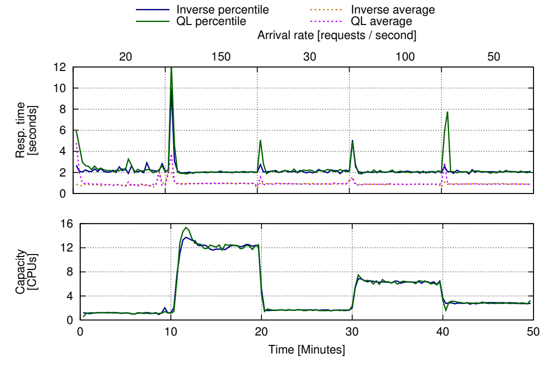
1) Statistical analysis: In this section, we present statistical results using cumulative distribution function to show the proportion of requests served against their perceived response times for the different configurations presented in Section V-C. For any configuration, we are interested in the user-perceived response time represented as an empirical cumulative distribution function for the total number of requests served. To show the response time value below which the proportion of the total requests are served, a line is drawn from the target percentile value to the target response time value stipulated in the SLA.

Table I show the cumulative distribution functions for the Wikipedia and FIFA workload experiments depicted under Figs. 3 and 4 respectively. In general, the aggregate behavior of both performance models behaves as expected. Meaning, the percentage of requests served below the target response time was close to the configured percentile value. However, close examination of the results reveal that under the inverse model with target percentile of 95%, 95% of the requests were almost exactly served below the corresponding target response time values under both the Wikipedia and FIFA workloads. For the 99% target under inverse model, 99% of the requests were served slightly further from the target response time values set under both workloads. However, under the queue length tail model, these values are a bit further for 95% target and relatively far for 99%. The aggregate behavior shows that the inverse tail model is better than the queue length model asserting the observation made under Section V-C.

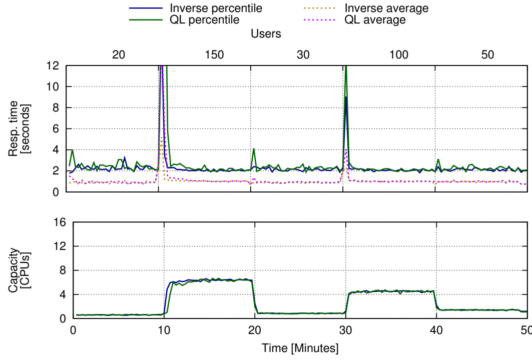
2) Control theoretic analysis: In this section we analyse cumulative behaviors such as the aggregate errors over



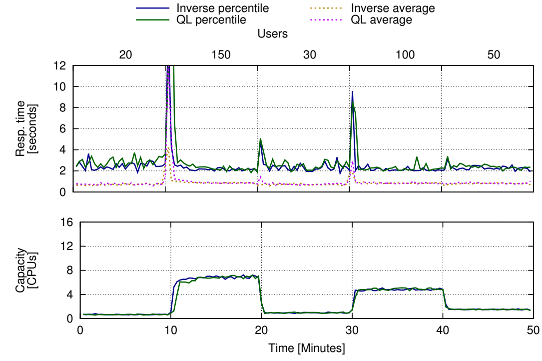
(a) open system model with 95% target



(b) open system model with 99% target



(c) closed system model with 95% target



(d) closed system model with 99% target

Figure 7: RUBiS–under open and closed system models with 2.0s target response time.

the span of the experiment (Section V-D) control theoretic techniques. 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:

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

$$IAE = \sum |e(t)|, \tag{7}$$

The error $e(t)$ is defined as the difference between the measured and target values at each control interval t.

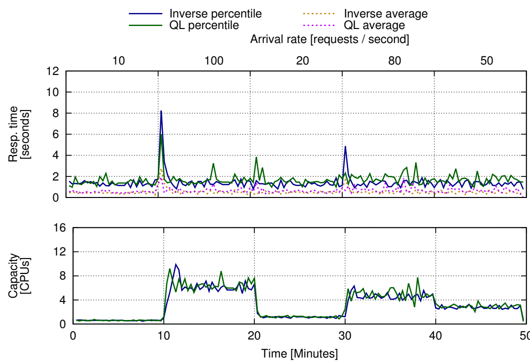
Table II shows the aggregate errors of both performance models for the RUBiS application with different target response times under the real workload. Under both the Wikipedia and FIFA workloads, both the ISE and IAE for the queue length tail model are higher than those for the inverse tail model for both 95% and 99% respectively. Besides, the ISE and IAE error values for 95% are lower than those for 99%. The implications of the ISE and IAE values are: (1) the inverse tail model performs better than the queue length tail model; (2) the two performance model performs better for lower percentile target (e.g., 95%) than higher target(e.g., 99%) under comparable workloads.

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

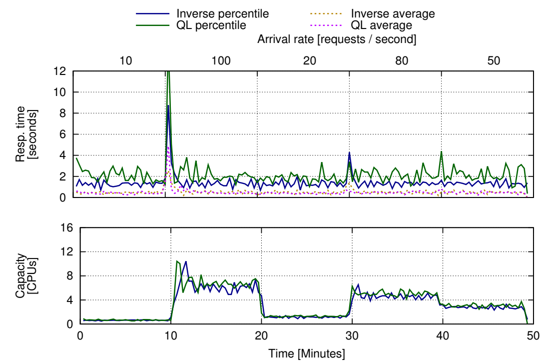
Application	**Workload**	**Perf. Model**	Target %	**ISE**	**IAE**
RUBiS	Wikipedia	Inverse	95	10.64	78.01
			99	32.33	163.82
		Queue length	95	48.00	195.82
			99	69.57	188.76
	FIFA	Inverse	95	1705.022	1511.10
			99	2294.00	1738.20
		Queue length	95	1876.20	1586.60
			99	2784.53	1909.80

E. Discussion

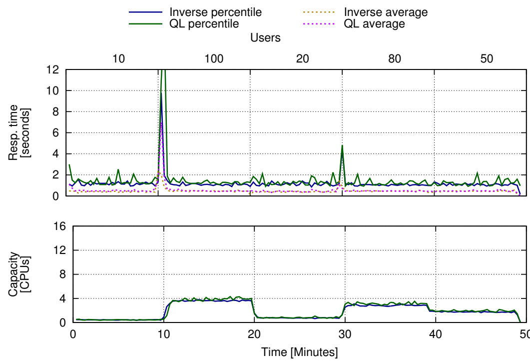
Experiments highlighted that both performance models behave as intended for all the configurations. Specifically, both performance models were capable of predicting the



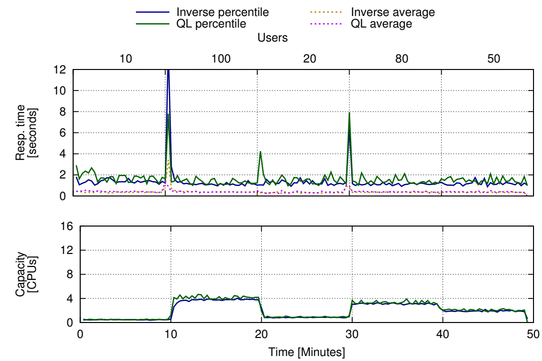
(a) open system model with 95% target



(b) open system model with 99% target



(c) closed system model with 95% target



(d) closed system model with 99% target

Figure 8: RUBiS–under open and closed system models with 1.0s target response time.

right amount of capacity required to satisfy the demands under both predictable Wikipedia-like and unpredictable FIFA-like workloads as well as under a variety of synthetic workloads generated based on open- and closed-system models. The implication is that the performance models can be used for different workload types.

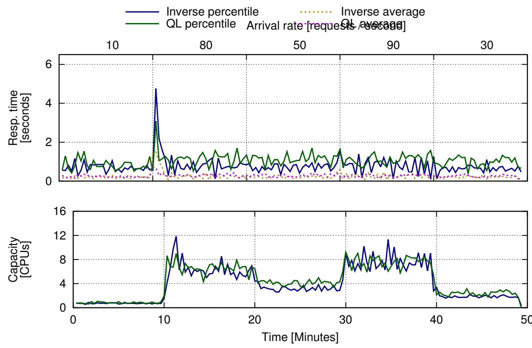
Besides reacting well to external changes, both performance models behaved well in response to applications' internal behavior changes. The performance models were able to adapt only by observing the applications' performance, without needing to be explicitly notified about such internal changes. This feature is very import since a new version of the application can be dynamically deployed, without requiring a lengthy retraining of the performance model, as demanded by the lean start-up movement and agile development practices.

The time series as well as the aggregate results show the inverse tail model performs better than the queue length tail model. More specifically, our results yielded the following **key findings**:

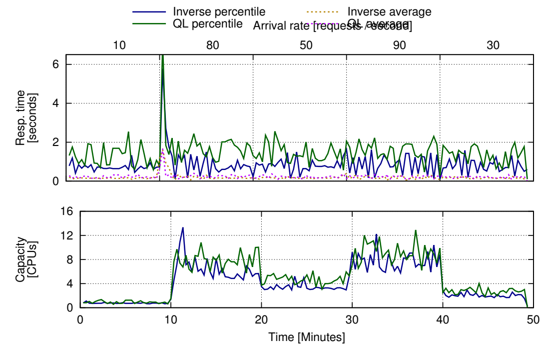
1) Both models are agnostic not only to dynamic changes in workload but also runtime changes in the application behavior such as applications' upgrades.
2) Both performance models are more stable when the response time target is relatively high. However, the inverse tail model is more stable than the queue-length tail model for lower targets.
3) Higher percentiles are more difficult and costly to maintain. For both performance models, the response time was kept more stable for 95-th percentile than 99-th percentile, and 99-th percentile requires more capacity than 95-th percentile for similar workloads.
4) Both models reach stability very quickly (i.e. within 40 seconds) after detecting a change in the system.
5) The inverse tail model is more precise and stable than the queue length tail model under all the configurations.

VI. Conclusion

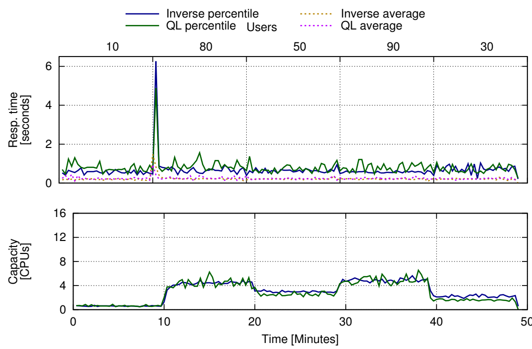
We have presented two novel tail response time performance models that map performance to capacity in order to



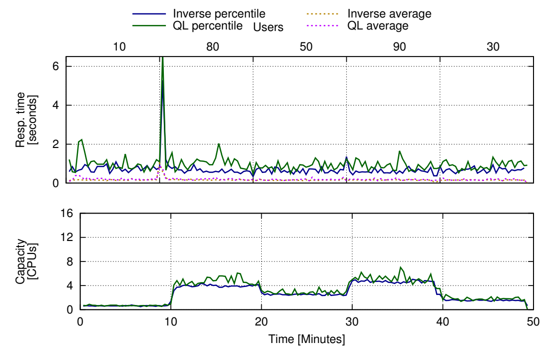
(a) open system model with 95% target



(b) open system model with 99% target

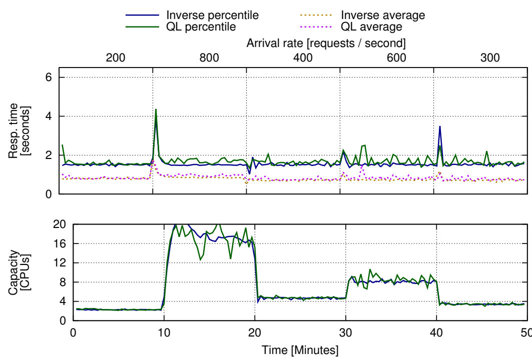


(c) closed system model with 95% target

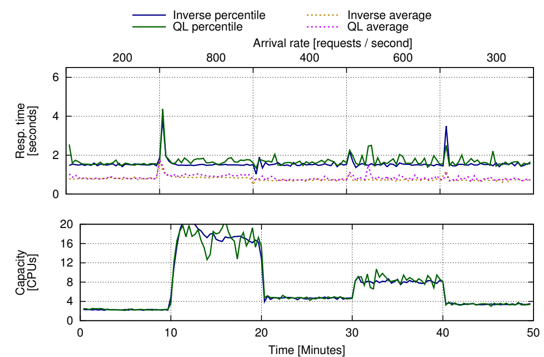


(d) closed system model with 99% target

Figure 9: RUBiS–under open and closed system models with 0.5s target response time.

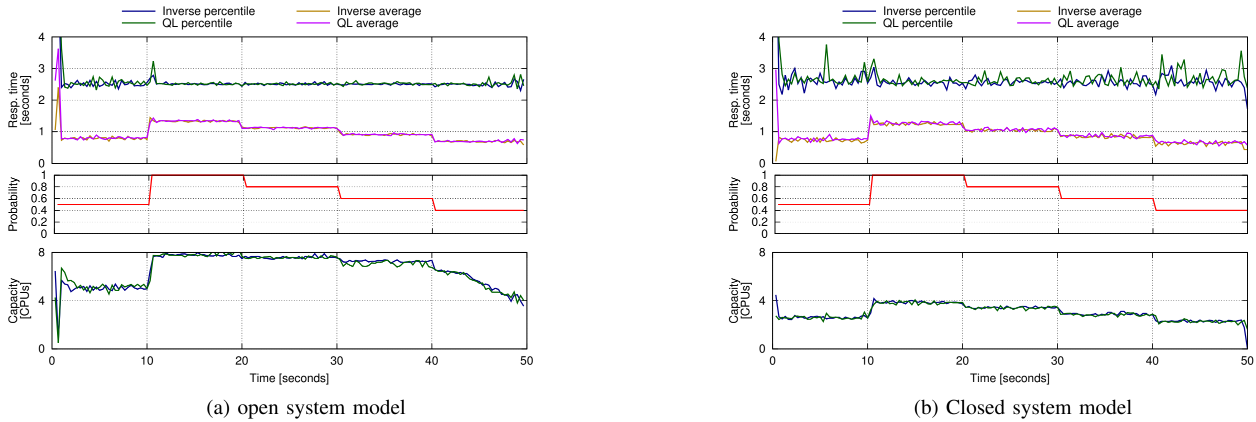


(a) open system model



(b) Closed system model

Figure 10: RUBBoS– under open and closed system models with 95% at 1.5s.



(a) open system model

(b) Closed system model

Figure 11: RUBiS–with constant arrival rate [number of requests] for 2.5s target response time with 95% while the probability value is changing.

Table I: CDFs of the two tail models Under Wikipedia and FIFA workloads.

		RUBiS		RUBBOS	
		95%	99%	95%	99%
Wikipedia	Inverse				
	QL				
FIFA	Inverse				
	QL				

provide performance guarantees for interactive applications deployed in the cloud. The models were able to allocate just the right amount of capacity required to meet the response time targets expressed in percentile such as 95% or 99% avoiding both capacity over- and under-provisioning.

Both models were evaluated in an extensive set of experiments using different applications with real workloads and synthetic 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 under both more predictable and unpredictable real workloads and synthetic workloads generated using open- and closed-system models . However, the inverse tail model is more stable than the queue length tail model. 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 modeling more resources such as memory and network and extending it to handle applications running across multiple PMs.

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] Marissa mayer of google: Speed good, ajax not so good, 2006. Available online: http://www.montparnas.com/articles/marissa-mayer-of-google-speed-good-ajax-not-so-good/, Visited 2015-04-06.

[2] Amazon found every 100ms of latency cost them 1 Available online: http://blog.gigaspaces.com/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/, Visited 2015-04-06.

[3] Second-generation cloud computing–iaas services what it means, and why we need it now, 2013. Available online: http://info.profitbricks.com/rs/profitbricks/images/Neovise-White-Paper-ProfitBricks-Second-Generation-Cloud-Computing-IaaS.pdf.

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

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

[6] Page view statistics for wikimedia projects, 2014. Available online: http://dumps.wikimedia.org/other/pagecounts-raw/, Visited 2014-10-20.

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

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

[9] Worldcup98, 2014. Available online: http://ita.ee.lbl.gov/html/contrib/WorldCup.html, Visited 2014-10-20.

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

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

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

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

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

[15] G. Galante and L.C.E. de Bona. A survey on cloud computing elasticity. In *Utility and Cloud Computing (UCC), 2012 IEEE Fifth International Conference on*, pages 263–270, Nov 2012.

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

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

[18] Wayne D. Gray and Deborah A. Boehm-Davis. Milliseconds matter: An introduction to microstrategies and to their use in describing and predicting interactive behavior, December 2000.

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

[20] Ewnetu Bayuh Lakew et al. Towards faster response time models for vertical elasticity. In *The 6th Cloud Control Workshop, part of the Proceedings of the 2014 IEEE Conference on Utility and Cloud Computing (UCC 2014)*, pages 560–565, 2014.

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

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

[23] Peter Mell and Timothy Grance. The NIST definition of cloud computing. Technical Report 800-145, National Institute of Standards and Technology (NIST), 2011.

[24] 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.

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

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

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

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

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

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

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

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