On the Value of Service Demand Estimation for Auto-Scaling

André Bauer, Johannes Grohmann, Nikolas Herbst, and Samuel Kounev

University of Würzburg, Würzburg, Germany,
`<firstname>.<lastname>@uni-wuerzburg.de`,
Home page: `https://descartes.tools`

Abstract. In the context of performance models, service demands are key model parameters capturing the average time individual requests of different workload classes are actively processed. In a system under load, due to measurement interference, service demands normally cannot be measured directly, however, a number of estimation approaches exist based on high-level performance metrics. In this paper, we show that service demands provide significant benefits for implementing modern auto-scalers. Auto-scaling describes the process of dynamically adjusting the number of allocated virtual resources (e.g., virtual machines) in a data center according to the incoming workload. We demonstrate that even a simple auto-scaler that leverages information about service demands significantly outperforms auto-scalers solely based on CPU utilization measurements. This is shown by testing two approaches in three different scenarios. Our results show that the service demand-based auto-scaler outperforms the CPU utilization-based one in all scenarios. Our results encourage further research on the application of service demand estimates for resource management in data centers.

Keywords: service demand estimation, auto-scaling, online estimation, elastic cloud computing

1 Introduction

The cloud computing paradigm has a high impact in the ICT domain as it allows on-demand access to data center resources (e.g., networks, servers, storage and applications). In order to guarantee a reliably operating service, mission-critical applications usually run with a fixed amount of resources. This has some drawbacks: On the one hand, if the application is not fully used, the resource consumption becomes inefficient; on the other hand, if an unexpected event occurs, the performance and availability is not ensured.

The domain of auto-scaling is concerned with automatically and precisely allocating the required amount of resources (e.g., number of VMs) for a given time period and changing this allocation as the demand changes. Existing auto-scaling mechanisms either assume as input an estimate on the processing speed of the resources or rely on measured resource utilization averages (r.t., Section 5).

In this work, we compare two auto-scaling approaches with identical underlying decision logic under changing workloads. The first approach uses the CPU utilization to scale the system, while the second one uses online service demand estimation. Both approaches scale three different application types: (i) a scalable application limited in performance mainly by hardware contention, (ii) a second application that has software bottlenecks leading to software contention, and (iii) a third application that exhibits both hardware and software contention. Given these three application types and the two different approaches to realize auto-scaling, we pose ourselves the following research questions:
RQ1: How can CPU utilization-based and service demand-based auto-scaling mechanisms be compared in a fair manner?
RQ2: In which scenarios does service demand-based auto-scaling outperform a CPU utilization-based mechanism?
RQ3: What are benefits of using service demand estimates instead of utilization measurements for automatic scaling decisions?

We structure the three main contributions of this paper as follows: We address RQ1 by discussing the competing auto-scaling approaches in Section 2. Then, the experiment setup is introduced. Afterwards, RQ2 and RQ3 are answered by comparing and quantifying the two approaches in three different scenarios. In Section 5, we summarize related work before concluding the paper.

2 Service Demand Estimation for Auto-Scaling

In this section, the foundations of the service demand estimation are explained. Afterwards, the competing auto-scaling approaches are introduced.

2.1 Service Demand Estimation

Service demands are a key parameter of stochastic performance models. A service demand is the average time a unit of work (e.g., request or transaction) spends obtaining service from a resource (e.g., CPU or hard disk) in a system, over all visits at the resource, excluding any waiting times [1, 2]. Different requests can be grouped into different *workload classes*. Service demands are normally considered on a per workload class basis.

In most realistic systems, the direct measurement of service demands is not feasible during operation [3] due to instrumentation overheads and possibly measurement interference. Willneker et al. [4] show that statistical estimation approaches can provide a comparable accuracy to direct measurements. Thus, we focus on statistical approaches for estimating the service demands.

The advantage of statistical estimation approaches compared to direct measurement techniques is their general applicability and low overheads. Estimation approaches typically rely only on coarse-grained measurements from the system (e.g., CPU utilization and end-to-end average response times) that can be obtained easily with monitoring tools without the need for fine-grained code instrumentation. These measurements are routinely collected for many applications (e.g., in data centers) and therefore estimation approaches are also applicable on systems serving production workloads.

Over the years, a number of approaches to service demand estimation have been proposed based on different statistical estimation techniques (e.g., linear regression [5, 6] or Kalman filters [7, 8]) and combined with laws from queueing theory. We refer to Spinner et al. [3] for an overview as well as a classification and experimental evaluation of the different approaches for service demand estimation.

LibReDE: For estimating the service demands in an online setting, our approach uses the Library for Resource Demand Estimation (LibReDE) [9]. LibReDE[1] is a library of ready-to-use implementations of state-of-the-art approaches to service demand estimation that can be used for online and offline analysis. It supports several different approaches to service demand estimation.

For the sake of simplicity, here we restrict ourselves to using only one approach, based on the Service Demand Law [6]:

The service demand law [2] states that the the service demand $D_{i,c}$ of c at resource i can be obtained by dividing the utilization $U_{i,c}$ of a resource i due to workload class c by the system throughput $X_{0,c}$ of workload class c:

$$D_{i,c} = \frac{U_{i,c}}{X_{0,c}}. \tag{1}$$

However, usually $U_{i,c}$ is not measurable, since we can only measure the total utilization of resource i caused by all workload classes running on i. Therefore, we have to partition the utilization among the different workload classes. Menascé et al. [2] and Lazowska et al. [1] solve this by collecting additional per-class data, while Brosig et al. [6] estimate it based on the response times of the different classes. Therefore, we only need to measure the average CPU utilization and the throughput of each workload class in order to estimate the service demands.

2.2 Competing Methods

In order to investigate the value of service demand estimation for auto-scaling, we compare the same threshold-based auto-scaling algorithm, as implemented for example on Amazon Web Services (AWS) EC2, but feed it with two different input parameters: (i) the measured average CPU utilization and (ii) the average system utilization based on queueing theory (see Equation 1). The main advantage of using the CPU utilization is that it is easy to measure, however, the thresholds for the scaling have to be tuned for each application individually and measurable load levels are limited at 100%. In contrast, the average system utilization based on queueing theory has no limitation and the thresholds can be defined independent of the application. However, the determination of the average system utilization requires application level metrics like response times

[1] LibReDE: https://descartes.tools/librede/

and trough put per request class. The thresholds can be either learned leveraging machine learning approaches or as in our case determined based on experience.

Taking AWS as example, per default they provide metrics such as CPU utilization, disk IO (disk read operations or disk write operation), and network IO for the auto-scaling decision. The reasons for choosing CPU utilization as scaling indicator are threefold: (i) in many cases the bottleneck resource may not known at configuration time, (ii) our software contention scenario is limited by the number of unlocked files and not by an IO rate, and finally, (iii) when using IO metrics, a deep knowledge of the IO characteristics of the machine is required, which appears unfeasible for cloud deployments.

The auto-scaling mechanism communicates with the cloud every minute and gathers VM specific information, such as the amount of running VMs and the average CPU utilization, and application specific information, such as request arrival rates. Algorithm 1 illustrates the simplified decision logic that forms the basis for both approaches. Important parameters for decision making are *up_threshold* and *down_threshold*. Input parameters are the current average system utilization $\overline{\rho}$ and the number of currently running VMs vms. In contrast to the CPU utilization-based approach where $\overline{\rho}$ is equal to the measured CPU load, the service demand-based approach uses the average system utilization based on the service demand law from queueing theory that offers a good trade-off between estimation time and accuracy [10]. The system utilization ρ is derived from the the arrival rate multiplied with the estimated service demand respectively the highest service demand if multi-class systems are scaled [11]. The service demand estimate is updated online every 10 minutes, as service demand estimated are expected to not change significantly in a short period of time.

In the second line, Algorithm 1 checks if the average system utilization $\overline{\rho}$ per VM is greater than a predefined threshold. While this condition is true, the new average system utilization per VM is calculated after iteratively increasing the number of VMs. Otherwise, if the average system utilization per VM is less than a predefined threshold, the number of VMs is decreased iteratively until $\overline{\rho}$ is greater than the threshold. Finally, the algorithm returns the number of VMs that are required (amount > 0) or that can be released (amount < 0).

3 Experiment Description

In order to obtain a authentic workload with a time-varying behavior, we use the Retailrocket[2] trace. This trace represents HTTP requests to servers of an anonymous real-world e-commerce website during June 2015. For our experiments, we sample the arrival rates every 15 minutes, i.e., each day consists of 96 data points. Furthermore, we crop out two days and speed-up the trace by the factor 15 so that each data point represents one minute.

The auto-scaling mechanisms are configured to monitor and auto-scale three different applications: (i) an application with its performance limited by hardware contention (CPU), (ii) an application exhibits software contention, and

[2] Retailrocket Source: https://www.kaggle.com/retailrocket/ecommerce-dataset

ALGORITHM 1: Decision logic.

```
Compute-Optimal-VMs (double $\overline{\rho}$, int vms)
    amount = 0;
    if $\overline{\rho}$ > up_threshold then // is $\overline{\rho}$ > a predefined threshold
        while $\overline{\rho}$ > up_threshold do
            amount++;
            $\overline{\rho} = \overline{\rho}$ * (vms / (vms + amount)); // calculates the new average utilization
    else if $\overline{\rho}$ <= down_threshold then // is $\overline{\rho} \leq$ a predefined threshold
        while $\overline{\rho}$ <= down_threshold do
            amount−−;
            $\overline{\rho} = \overline{\rho}$ * (vms / (vms + amount)); // calculates the new average utilization
            if $\overline{\rho}$ > up_threshold then
                amount++; // undo
    return amount;
```

(iii) an application that exhibits both hardware and software contention. The first application is a CPU-intensive Java Enterprise application that is a re-implementation of the LU worklet from SPEC's Server Efficiency Rating Tool SERT^{TM}2. The application calculates the LU Decomposition [12] of a random generated $n \times n$ matrix, where n is the GET parameter of each HTTP request.

The second application is also a Java Enterprise application. This application is used by a content-delivery service provider with limited capacities. Here, the incoming HTTP requests try to read a file out of a limited pool of randomly generated files. While a file is being read by an HTTP request, it is locked and cannot be accessed by other requests, i.e., an incoming HTTP-request either successfully reads a file or waits until a file is unlocked.

The third scenario contains the constraints of both scenarios. At first the application calculates the LU decomposition. Afterwards, it tries to read a file out of the limited pool.

All applications are deployed on WildFly application servers in our private cloud infrastructure. Here, we use an Apache CloudStack[3] cloud that manages virtualized Xen-Server hosts. This cloud environment is running in a cluster of 11 homogeneous HP servers. Eight of them are managed by CloudStack. The last three servers are not part of the cloud and are used for hosting the software for the cloud management as well as the benchmark framework: (i) the load-balancer (Citrix Netscaler[4]) and the cloud management for CloudStack, (ii) the auto-scaling mechanisms, and (iii) the load driver and the experiment controller. The specification of each physical machine can be found in Table 1.

Elasticity Benchmarking Framework: In order to evaluate the two approaches, we use the BUNGEE Cloud Elasticity Benchmark controller [13]. First, the controller constructs for the SuT (System under Test) a discrete mapping function that determines for each load intensity the associated minimum amount of resources required to meet the SLOs. Based on this mapping and a predefined

[3] Apache CloudStack: https://cloudstack.apache.org/

[4] Citrix Netscaler: https://www.citrix.de/products/netscaler-adc/

Table 1. Specification of the Servers.

Criteria	**Server**	**Worker VMs**
Model	HP DL160 Gen9	–
Operating System	Xen-Server	Centos 6.5
CPU	8 cores	2 vcores
Memory	32 GB	4 GB

workload profile, the SuT is stressed while BUNGEE monitors the supplied VMs. After the experiment, the elasticity and user-oriented metrics based on the collected monitoring data are calculated.

4 Results

This section presents the experiments and the respective findings. First, the specific metrics used to evaluate and compare the two approaches are defined. Then, the hardware contention scenario is discussed. In Section 4.3, the software contention scenario is examined. Afterwards, the mixed scenario is discussed. Finally, the results are discussed with their associated threats to validity.

4.1 Quantifying the Auto-Scaler Performance

In order to evaluate the two competing approaches, on the one hand, we use user-oriented metrics such as SLO (service level objective) violations in combination with the average response time. On the other hand, we use system-oriented elasticity metrics endorsed by the Research Group of the Standard Performance Evaluation Corporation (SPEC) [14]. In particular, we use the provisioning accuracy and the wrong provision time share. Using only single metrics, it is hard to gain insight into the performance differences between the two approaches. Hence, we use multiple metrics and summarize the gain of using each approach using an aggregate metric called elasticity speedup. Each elasticity metric is described in the remainder of this section. For the following equations, we define: (i) T as the experiment duration and time $t \in [0, T]$, (ii) s_t as the resource supply at time t, and (iii) d_t as the demanded resource units at time t. The demanded resource units d_t is the minimal amount of VMs required to meet the SLOs under the load intensity at time t. Δt denotes the time interval between the last and the current change either in demand d or supply s. The curve of demanded resource units d over time T is derived by BUNGEE, see Section 3. The resource supply s_t is the monitored number of running VMs at time t.

Provisioning accuracy θ_U and θ_O: The provisioning accuracy describes the relative amount of resources that are under-provisioned, respectively, over-provisioned during the measurement interval. In order words, the under-provisioning accuracy θ_U is the amount of missing resources normalized by the current demanded resource units that are required to meet the SLOs normalized by the experiment time. Similarly, the over-provisioning accuracy θ_O is the amount of resources that the auto-scaler supplies in excess. The range of this metric is the interval $[0, \infty)$, where 0 is the best value and indicates that the supply curve follows lays on demand curve during the entire measurement interval.

$$\theta_U[\%] := \frac{100}{T} \cdot \sum_{t=1}^{T} \frac{max(d_t - s_t, 0)}{d_t} \Delta t \tag{2}$$

$$\theta_O[\%] := \frac{100}{T} \cdot \sum_{t=1}^{T} \frac{max(s_t - d_t, 0)}{d_t} \Delta t \tag{3}$$

Wrong provisioning time share τ_U and τ_O: The wrong provisioning time share captures the time in which the system is an under-provisioned, respectively over-provisioned, state during the experiment interval, i.e., the under-provisioning time share τ_U is the time relative to the measurement duration, in which the system is under-provisioned. Similarly, the over-provisioning time share τ_O is the time relative to the measurement duration in which the system is over-provisioned. The range of this metric is the interval $[0, 100]$. The best value 0 is achieved, when the system has during the measurement no over- or under-provisioning.

$$\tau_U[\%] := \frac{100}{T} \cdot \sum_{t=1}^{T} max(sgn(d_t - s_t), 0) \Delta t \tag{4}$$

$$\tau_O[\%] := \frac{100}{T} \cdot \sum_{t=1}^{T} max(sgn(s_t - d_t), 0) \Delta t \tag{5}$$

Elastic Speedup ϵ This comparing method calculates for each approach its gain based on its scaling behavior compared to the no auto-scaling scenario. This approach allows to compare our two approach by taking only the system-oriented metrics into account. In other words, the elasticity metrics $x = (\theta_U, \theta_O, \tau_U, \tau_O)$ of each approach a is compared to the metrics of the no-auto scaling scenario n. To this end, the geometrical mean of the ratio between each metric pair is calculated. If the value is greater than 1, the proposed method is better than having no auto-scaler and the value reflects the gain. If the values is less than 1, the approach is worse than having no auto-scaler. Mathematically, the elastic speedup ϵ for an auto-scaler a based on the no auto-scaling scenario n can be formulated as:

$$\epsilon_n := \left(\frac{\theta_{U,n}}{\theta_{U,a}} \cdot \frac{\theta_{O,n}}{\theta_{O,a}} \cdot \frac{\tau_{U,n}}{\tau_{U,a}} \cdot \frac{\tau_{O,n}}{\tau_{O,a}} \right)^{\frac{1}{4}} \tag{6}$$

4.2 Hardware Contention Scenario

The results for the hardware contention scenario are depicted in Figure 1. This diagram is divided into three parts: The first part shows the scaling behavior, the second one the average system utilization for each approach, and the last

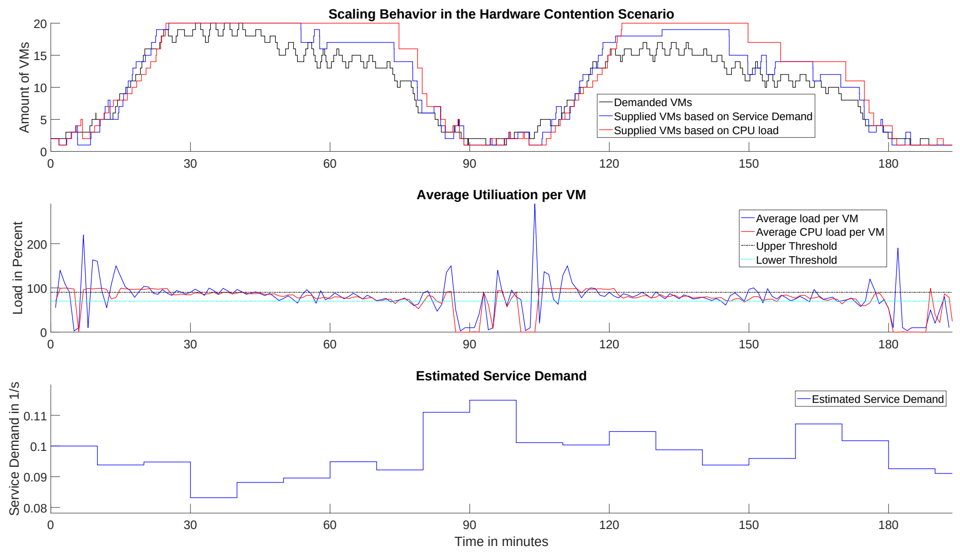

Fig. 1. Scaling behavior in the hardware contention scenario.

one the estimated service demand. In the first part, the black line describes the curve of demanded resource units (determined by BUNGEE, see Section 3), the red line the scaling behavior for the CPU utilization-based approach, and the blue line the scaling behavior for the service demand-based approach. Here, both approaches tend to over-provision the system during the decreasing tail of each day. However, the auto-scaler based on CPU utilization has more instances over-provisioned during this period compared to the service demand-based approach. Furthermore, the service demand-based auto-scaler can handle the increasing load during each day more efficiently than the CPU utilization-based one.

Table 2. Results for the hardware contention scenario.

Metric	**No Scaling**	**Service demand-based**	**CPU utilization-based**
θ_U (accuracy$_U$)	**3.20**%	7.54%	7.46%
θ_O (accuracy$_O$)	203.28%	**14.60**%	23.57%
τ_U (timeshare$_U$)	22.89%	**19.10**%	22.20%
τ_O (timeshare$_O$)	67.38%	**62.56**%	62.65%
ϵ (Elastic Speedup)	1.00	**1.66**	1.42
ψ (SLO violations)	45.72%	**8.40**%	12.67%
Avg. #VMs	15.00	8.58	7.93
Avg. response time	2.62 s	**0.70** s	0.94 s
Med. response time	2.86 s	**0.22** s	0.24 s

These observations can be explained by looking at the middle part of the figure where the blue line shows the average system utilization and the red line the average CPU utilization. The black dashed line represents the threshold for up-scaling (90%) and the cyan dashed line the threshold for down-scaling (70%). While the CPU utilization is limited by 100%, the system utilization can have values higher than this limit, for instance, at minute 115 where the system is in under-provisioned state for both approaches, the CPU utilization is 100% and the system utilization is 160%. The service demand-based approach assigns 3 VMs to handle this utilization and the CPU utilization-based one only 1 VM.

The estimated service demand (see Section 2.2) for calculating the system utilization, depicted in the bottom part of the figure, has the avgerage value of $0.099 \pm 0.016\frac{1}{s}$.

To enable quantitative comparison, we calculated the elasticity metrics (see Section 4.1) as well as some user-oriented metrics listed in Table 2. Here, each row shows a metric and each column an auto-scaler. The best values are printed in bold. As a baseline scenario ("no auto-scaling"), we run 15 VMs (75% of the available VMs) throughout the experiment duration. When comparing only the individual elasticity metrics, the service demand-based approach exhibits the best results for 3 out of 4 metrics. Thus, it also achieves the highest elastic speedup (1.66). The CPU utilization-based approach also has an elastic speedup greater than 1, i.e., both approaches are more efficient than the baseline (no auto-scaling) scenario. Furthermore, both approaches use less VMs, achieve significantly higher SLO conformance, and have a lower response time than the no auto-scaling scenario. However, the service demand-based auto-scaler has the lowest amount of SLO violations (8.40%) and the lowest response time (0.70 s).

4.3 Software Contention Scenario

Similar to Section 4.2, the results for the software contention scenario are depicted in Figure 2. The scaling behavior of both approaches is shown in the upper diagram, where the black line describes the curve of demanded resource units, the red line the scaling behavior for the CPU utilization-based approach, and the blue line the scaling behavior for the service demand-based approach. In contrast to the hardware contention scenario, this scenario marginally stresses the CPU. Although the thresholds for up-scaling (30%) and down-scaling (5%) are adjusted, the CPU utilization-based approach is not able to meet the required VMs during the two days. The supply curve of the service demand-based approach is close to the demand curve. There are only few intervals in which the system is in an under-provisioned state for a short time. In analogy to the hardware contention scenario, the system utilization is more suitable to describe the required amount of VMs. In both scenarios the service demand-based approach uses the same thresholds in contrast to the CPU utilization-based one.

The estimated service demand (see Section 2.2) for calculating the system utilization, depicted in the last row of the figure, has the average value of $0.153 \pm 0.025\frac{1}{s}$.

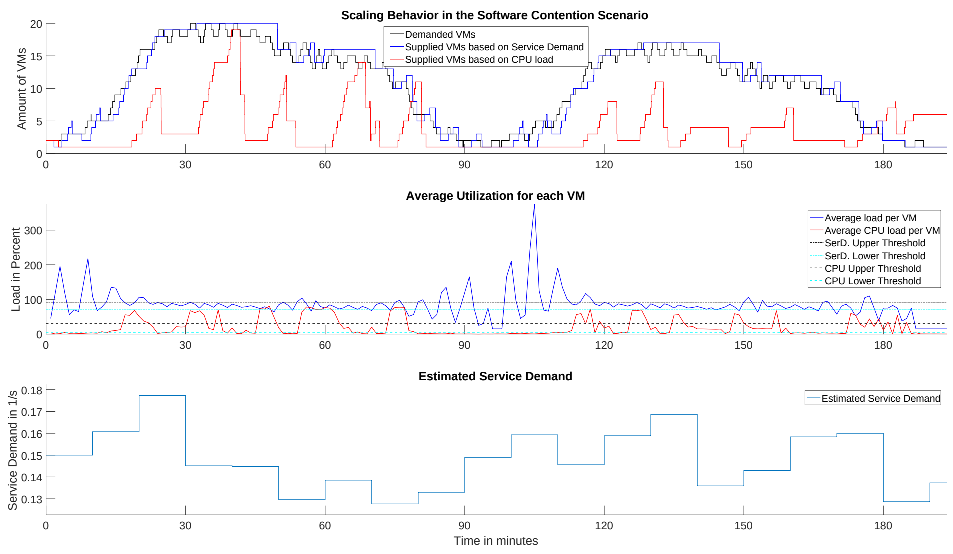

Fig. 2. Scaling behavior in the software contention scenario.

Table 3. Results for the software contention scenario.

Metric	**No Scaling**	**Service demand-based**	**CPU utilization-based**
θ_U (accuracy$_U$)	**3.20**%	7.64%	60.45%
θ_O (accuracy$_O$)	203.28%	**8.36**%	24.67%
τ_U (timeshare$_U$)	22.89%	**27.02**%	86.83%
τ_O (timeshare$_O$)	67.38%	43.37%	**9.29**%
ϵ (Elastic Speedup)	1.00	**1.91**	0.96
ψ (SLO violations)	8.64%	**6.27**%	97.25%
Avg. #VMs	15.00	8.21	6.69
Avg. response time	1.07 s	1.04 s	1.97 s
Med. response time	**0.98** s	**0.98** s	2.00 s

In order to compare the two approaches, we calculate the elasticity metrics, collect user information, and compare the scaling behavior with the baseline no (auto-scaling scenario) in which 15 VMs are permanently running during the experiment. The results are summarized in Table 3. Here, each row represents a metric and each column an auto-scaler. The best values are highlighted in bold. The service demand-based auto-scaler exhibits the best elastic speedup, the lowest amount of SLO violations, and the lowest response time. In contrast, the CPU utilization-based approach has an elastic speedup lower than 1, i.e., the performance judged by the elasticity metrics is worse than the no auto-scaling scenario. This can also be seen by the high amount of SLO violations (97.25%).

4.4 Mixed Contention Scenario

The results for the mixed contention scenario are depicted in Figure 3. Here, the scaling behavior of the service demand-based approach (blue curve), the scaling behavior of the CPU utilization-based approach (red curve), and the demanded resource units (black curve) are shown in the upper part of the figure. As this application has both software and hardware contention, we determined and calibrated the upper-threshold (55%) and lower-threshold (40%) for the CPU utilization-based approach. While this approach has problems to meet the required VMs at pitch of the first day, the remaining days are covered better than in the software contention scenario. Furthermore, there is less over-provisioning than in the hardware contention scenario. Similar to the previous scenarios, the service demand-based approach has almost no under-provisioning and tends to over-provision slightly.

The estimated service demand (see Section 2.2) for calculating the system utilization, depicted in the bottom row of the figure, has the average value of $0.238 \pm 0.043 \frac{1}{s}$.

The observed scaling behavior is quantified by the metrics shown in Table 4. Here, each row represents a metric and each column an auto-scaler. While, the service demand-based approach has the best under-provision timeshare metric, it also has the highest elastic speedup (1.71) and the lowest SLO violations (4.77%). Also the CPU utilization-based approach has a value higher than 1 and thus, both approaches are more efficient than the no auto-scaling scenario, in which 15 VMs are running throughout the experiment.

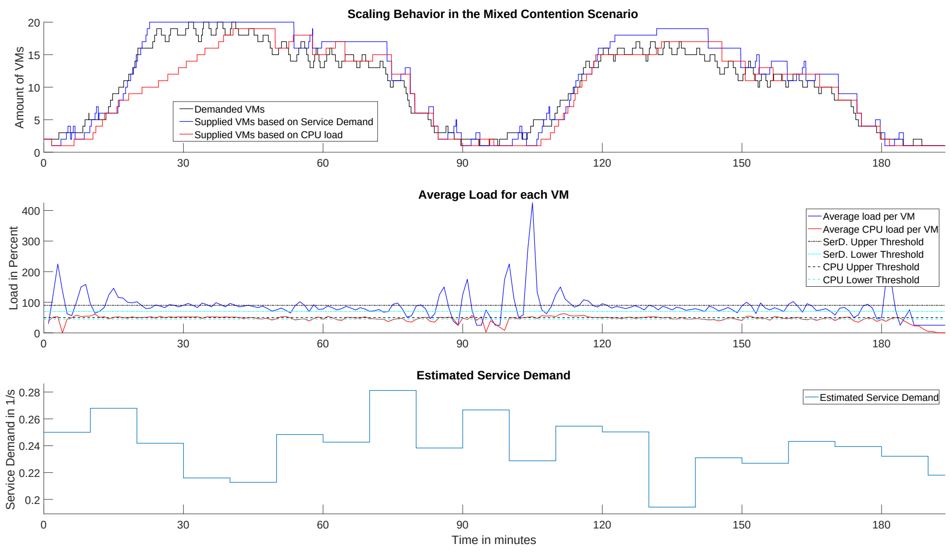

Fig. 3. Scaling behavior in the mixed contention scenario.

Table 4. Results for the mixed contention scenario.

Metric	**No Scaling**	**Service demand-based**	**CPU utilization-based**
θ_U (accuracy$_U$)	**3.20**%	7.13%	14.61%
θ_O (accuracy$_O$)	203.28%	14.12%	**7.05**%
τ_U (timeshare$_U$)	22.89%	**19.28**%	42.86%
τ_O (timeshare$_O$)	67.38%	59.90%	**35.03**%
ϵ (Elastic Speedup)	1.00	**1.71**	1.60
ψ (SLO violations)	6.40%	4.77%	11.96%
Avg. #VMs	15.00	8.96	9.51
Avg. response time	1.95 s	**1.94** s	1.95 s
Med. response time	**1.95** s	**1.95** s	**1.95** s

4.5 Summary

When comparing the different scenarios, the service demand-based auto-scaler behaves in a similar manner in each scenario and per day. In contrast, the CPU utilization-based approach behaves differently depending on the scenario. Furthermore, the service demand-based auto-scaler achieves the best values of the elasticity and user-oriented metrics. While the CPU utilization-based approach has problems in the software contention scenario, it is still more efficient than using no auto-scaling in the hardware contention and mixed contention scenarios. Note that the elasticity metrics of the no auto-scaling are identical per definition throughout all experiments.

Considering the applicability of both approaches, the CPU utilization-based auto-scaler is easy to setup and needs no further instrumentation. The CPU utilization can be gathered through standard tools or services such as SNMP (Simple Network Management Protocol). In contrast, the service demand-based approach requires a more complex, yet still not intrusive, instrumentation. Either the service demand has to be determined ahead of time assuming it is static or the service demand has to be estimated online as done in this paper. For this, the resource estimator needs structural application knowledge and information that may require a basic instrumentation of the application to monitor high-level metrics such as like request completion rates and response times.

To avoid the performance variability of public cloud infrastructures due to overbooking of resources and background-traffic, which also renders CPU utilization measures both unstable and unreliable [15], we ran the experiments in our private cloud environment under controlled conditions. Based on our experience with experiments in public clouds, CPU utilization based auto-scaling is supposed to perform worse than under controlled conditions while having a minor impact on the service demand-based approach.

We chose two similar days of the Retailrocket trace and conduct long experiments in order to validate the measurements internally respectively to have the second day as repetition of the first one. As the scaling behavior is influenced by the defined thresholds, we cannot prove that we have chosen the optimal ones.

5 Related Work

We studied two surveys on existing auto-scalers from Jennings and Stadler [16], and Lorido-Botran et al. [17]. Besides a broad overview of existing auto-scalers, the survey in [17] proposes a classification of auto-scalers into five groups: (i) threshold-based rules [18, 19], (ii) queueing theory [20, 21], (iii) control theory [22, 23], (iv) reinforcement learning [24, 25], and (v) time series analysis [26, 27]. While analyzing the auto-scalers in these survey, we can conclude that service demands - on a higher abstraction level the server's processing speed - is estimated indirectly and in an application-specific way for threshold-based rule approaches, assumed to be provided or calibrated as input for queueing and control theory auto-scaler, or learned over a training phase for reinforcement learning approaches. As examples for approaches leveraging time-series analysis, the auto-scaler called CloudScale, which is designed by Shen et al. [28], predicts the demand for resources with fast Fourier transformation (FFT) algorithms. A similar approach, called AGILE, is proposed by Nguyen et al. [29] who leverages wavelets instead. However, the demand for resources in the context of both of the above papers is understood as the CPU utilization. They sample the CPU utilization as time series and predict the future CPU load, instead of estimating the current processing speed of servers in terms of resource consumption per request.

To the best of our knowledge, there is only one auto-scaler proposed by Spinner et al. [30] that uses online service demand estimation to scale-out one web-server by adding virtual CPU cores. In contrast to this work, here we focus on horizontal scaling by adding virtual machine instances to a load-balancer. However, the results of Spinner et al. [30] support our message, as the results demonstrate an increased controller stability for an service-demand based approach compared a classical one based on CPU utilization threshold.

6 Conclusion and Discussion

In this paper, we compare two different approaches for auto-scaling: (i) based on measurements of the CPU utilization and (ii) based on service demand estimation as input values for an identical decision logic. To answer RQ1 "How can CPU utilization-based and service demand-based auto-scaling mechanisms be compared in a fair manner?", the two approaches scale three different types of applications: (i) a scalable application limited in performance mainly by hardware contention, (ii) a second application that has software bottlenecks, and (iii) a third application that exhibits both hardware and software contention. We use an established set of elasticity metrics to evaluate and compare the two auto-scaling approaches on a level playing field. We summarize our research findings as follows: The service demand-based approach is independent of the scenario, i.e., the service demand estimation does not rely on knowing the bottleneck resource and can be configured independently of the application. Furthermore, it achieves the best values of the various metrics in all scenarios and exhibits a similar scaling behaviour. These findings answer both RQ2 "In which scenarios does service demand-based auto-scaling outperform a CPU utilization-based mechanism?" and RQ3 "What are benefits of using service demand estimates instead of utilization measurements for automatic scaling decisions?".

We are confident that our results can encourage further research activity in the application of service demand estimation from the performance modeling domain for resource management in cloud data centers. For future work, we plan to extend the set scope of our analysis by conducting experiments in other cloud environments and investigating further types of applications. Additionally, a workload containing multiple workload classes will be considered. Finally, more research on different approaches to service demand estimation is planned, since the quality of the service demand estimates has direct influence on the auto-scaler decisions.

Acknowledgements

This work was co-funded by the German Research Foundation (DFG) under grant No. (KO 3445/11-1) and by Google Inc. (Faculty Research Award).
Arif Merchant, Google Inc., contributed with helpful ideas and feedback.

References

1. Lazowska, E.D., Zahorjan, J., Graham, G.S., Sevcik, K.C.: Quantitative system performance: computer system analysis using queueing network models. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1984)
2. Menascé, D.A., Dowdy, L.W., Almeida, V.A.F.: Performance by Design: Computer Capacity Planning By Example. Prentice Hall PTR, Upper Saddle River, NJ, USA (2004)
3. Spinner, S., Casale, G., Brosig, F., Kounev, S.: Evaluating Approaches to Resource Demand Estimation. Perform. Evaluation **92** (October 2015) 51 – 71
4. Willnecker, F., more: Comparing the Accuracy of Resource Demand Measurement and Estimation Techniques. In Beltrán, M., Knottenbelt, W., Bradley, J., eds.: EPEW 2015. Volume 9272 of Lecture Notes in Computer Science., Springer (August 2015) 115–129
5. Rolia, J., Vetland, V.: Parameter estimation for performance models of distributed application systems. In: CASCON '95, IBM Press (1995) 54
6. Brosig, F., Kounev, S., Krogmann, K.: Automated Extraction of Palladio Component Models from Running Enterprise Java Applications. In: VALUETOOLS '09. (2009) 1–10
7. Wang, W., more: Application-Level CPU Consumption Estimation: Towards Performance Isolation of Multi-tenancy Web Applications. In: IEEE CLOUD 2012. (June 2012) 439 –446
8. Zheng, T., Woodside, C., Litoiu, M.: Performance Model Estimation and Tracking Using Optimal Filters. IEEE TSE **34**(3) (May 2008) 391–406
9. Spinner, S., Casale, G., Zhu, X., Kounev, S.: Librede: A library for resource demand estimation. In: ACM/SPEC ICPE 2014, New York, NY, USA, ACM (2014) 227–228

10. Grohmann, J., Herbst, N., Spinner, S., Kounev, S.: Self-Tuning Resource Demand Estimation. In: Proceedings of the 14th IEEE International Conference on Autonomic Computing (ICAC 2017). (July 2017)
11. Bolch, G., more: Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications. John Wiley & Sons (2006)
12. Bunch, J.R., Hopcroft, J.E.: Triangular factorization and inversion by fast matrix multiplication. Mathematics of Computation **28**(125) (1974) 231–236
13. Herbst, N., Kounev, S., Weber, A., Groenda, H.: BUNGEE: An Elasticity Benchmark for Self-Adaptive IaaS Cloud Environments. In: SEAMS 2015, IEEE Press (2015) 46–56
14. Herbst, N., more: Ready for Rain? A View from SPEC Research on the Future of Cloud Metrics. CoRR **abs/1604.03470** (2016)
15. Iosup, A., Yigitbasi, N., Epema, D.: On the Performance Variability of Production Cloud Services. In: CCGrid 2011. (2011) 104–113
16. Jennings, B., Stadler, R.: Resource Management in Clouds: Survey and Research Challenges. Journal of Network and Systems Management **23**(3) (2015) 567–619
17. Lorido-Botran, T., Miguel-Alonso, J., Lozano, J.A.: A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments. Journal of Grid Computing **12**(4) (2014) 559–592
18. Han, R., Guo, L., more: Lightweight Resource Scaling for Cloud Applications. In: IEEE/ACM CCGrid 2012, IEEE (2012) 644–651
19. Maurer, M., Brandic, I., Sakellariou, R.: Enacting Slas in Clouds Using Rules. In: Euro-Par 2011 Parallel Processing. Springer (2011) 455–466
20. Urgaonkar, B., more: Agile Dynamic Provisioning of Multi-tier Internet Applications. ACM TAAS **3**(1) (2008) 1
21. Zhang, Q., Cherkasova, L., Smirni, E.: A Regression-based Analytic Model for Dynamic Resource Provisioning of Multi-tier Applications. In: IEEE ICAC 2007, IEEE (2007) 27–27
22. Kalyvianaki, E., Charalambous, T., Hand, S.: Self-adaptive and Self-configured CPU Resource Provisioning for Virtualized Servers Using Kalman Filters. In: ACM ICAC 2009, ACM (2009) 117–126
23. Ali-Eldin, A., Tordsson, J., Elmroth, E.: An Adaptive Hybrid Elasticity Controller for Cloud Infrastructures. In: IEEE NOMS 2012, IEEE 204–212
24. Tesauro, G., Jong, N.K., Das, R., Bennani, M.N.: A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation. In: IEEE ICAC 2006, IEEE (2006) 65–73
25. Rao, J., more: VCONF: a Reinforcement Learning Approach to Virtual Machines Auto-configuration. In: ACM ICAC 2009, ACM 137–146
26. Iqbal, W., Dailey, M.N., Carrera, D., Janecek, P.: Adaptive Resource Provisioning for Read Intensive Multi-tier Applications in the Cloud. Future Generation Computer Systems **27**(6) (2011) 871–879
27. Chen, G., more: Energy-Aware Server Provisioning and Load Dispatching for Connection-Intensive Internet Services. In: NSDI. Volume 8. (2008) 337–350
28. Shen, Z., Subbiah, S., Gu, X., Wilkes, J.: Cloudscale: elastic resource scaling for multi-tenant cloud systems. In: ACM Symposium on Cloud Computing, ACM (2011)
29. Nguyen, H., more: Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In: ICAC. Volume 13. (2013) 69–82
30. Spinner, S., more: Runtime vertical scaling of virtualized applications via online model estimation. In: IEEE SASO 2014, IEEE (2014) 157–166