CEDULE: A Scheduling Framework for Burstable Performance in Cloud Computing

Ahsan Ali[1], Riccardo Pinciroli[2], Feng Yan[1], and Evgenia Smirni[2]

[1]University of Nevada, Reno, Reno, NV, USA, {aali,fyan}@unr.edu
[2]College of William and Mary, Williamsburg, VA, USA, {rpinciro,esmirni}@cs.wm.edu

***Abstract*—Cloud service providers use the concept of "burstable performance instance" that can temporally ramp up its performance to handle bursty workloads by utilizing spare resources. The state-of-the-practice to using the available burst capacity is independent of the workload, which results in squandering spare resources. In this work, we quantify and optimize the efficiency of using burst capacity so that it benefits both cloud service providers and end users. More specifically, we use a throttling mechanism as a control knob to continuously adapt the amount of spare resources based on workload characteristics such as traffic intensity. To identify optimal throttling, we integrate lightweight profiling and quantile regression in a synergistic way and build a prediction model that accurately predicts tail latency. We build an autonomic scheduling framework called CEDULE that can make adaptive scheduling decisions to maximize the efficiency of spare resources while achieving user defined SLOs. We conduct extensive experimental evaluations of the proposed scheduling framework on Amazon EC2 using popular benchmark applications, such as Sysbench, YCSB, and TPC-W. Experimental results demonstrate the high accuracy of the prediction model, i.e., average errors range from 1% to 15%. The effectiveness of CEDULE is verified as it can triple the efficiency of spare resources while meeting stringent SLOs.**

I. Introduction

Cloud computing is adopted by corporations and individuals for its flexible, reliable, and cost-effective services [5]. Cloud providers make their services available through virtual machines (or instances) of different capacities to better serve different user needs. The cost of an instance depends on its capacity. Typically, the less powerful a virtual machine, the lower its cost [23]. Recently, cloud providers [26, 27, 29] have introduced the concept of *burstable performance* instance, a new type of low-cost instance that has a guaranteed performance base but that can burst to significant better performance for certain amounts of time. Burstable instances are used for applications (e.g., micro-services and small and medium databases [26]) that usually do not need consistently high computational power, but may require higher computational power from time to time to deal with a burst of heavy load over a short period of time. In this case, the instance can ramp up its CPU performance for a limited time to effectively process the increased amount of requests. Burstable instances are the cheapest instances currently available [23], e.g., the monthly price for the smallest Amazon's on-demand instance (i.e., t2.nano) is almost 17 times lower than the monthly price of the smallest non-bursting instance (i.e., m5.large).

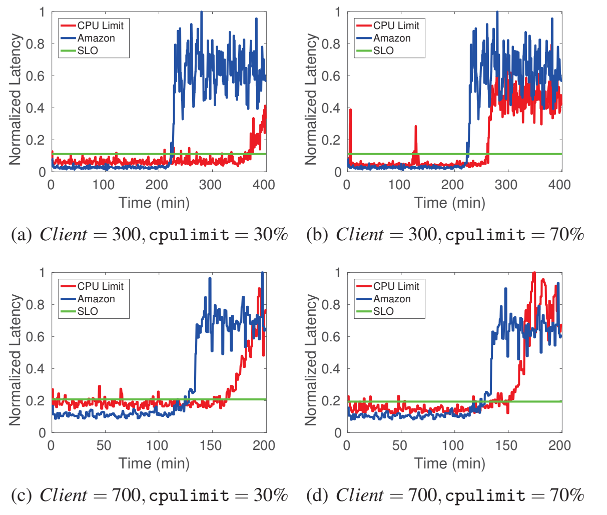
In this paper we use the Amazon EC2 platform and its t2 instances (i.e., burstable instances) for a case study. A t2 instance is created with a specific amount of initial CPU credits, this amount depends on the instance size. One credit provides the maximum computational power of a CPU core for a minute (i.e., the CPU is utilized at 100%). If CPU is less utilized, then its credit consumption reduces. For example, one credit is consumed in two minutes if the CPU runs at 50% utilization. After all initial credits are depleted, the instance operates at the baseline performance but periodically generates new credits with a rate that is commensurate to the instance's size, i.e., the credit generation is capped. In addition, the amount of credits generated in one hour is equal to the number of credits needed for the CPU to run with baseline performance for the same amount of time. Therefore, the average credit level remains unchanged. Finally, all credits (except for the initial ones) expire if they are not used for 24 hours since their generation. Table I summarizes the main characteristics of t2 instances.

TABLE I: Performance characteristics of four t2 instances: t2.nano, t2.micro, t2.small, and t2.medium. Note that the baseline performance refers to each available vCPU.

T2 size	vCPU	Init. Cr.	Gen. rate[cr/hr]	Baseln. Perf.	Max Cr.
nano	1	30	3	5%	72
micro	1	30	6	10%	144
small	1	30	12	20%	288
medium	2	60	24	20%	576

The capacity of burstable performance instances is enabled by spare resources, this is why Cloud service providers can offer such low prices for burstable instances. A more efficient way of using spare resources can greatly benefit cloud service providers as the same amount of spare resources can be multiplexed across more users but at the same time can also benefit end users by significantly reducing cost as otherwise a more expensive instance may need to be used for achieving the users's SLO. In this paper, we investigate the effectiveness of the state-of-the-practice for using burstable capacity/spare resources, our target is to improve the efficiency of using burstable instances. We broadly define efficiency as the amount of work that can be done using the burstable capacity of the spare resource, the more work is done, the higher its efficiency. To quantify "burstable efficiency", we use as metric the credit depletion time. Past work [24] showed that it is possible to extend the initial credit deletion period using the `cpulimit` utility [28] that limits the CPU usage of a process. Using `cpulimit` effectively forces CPU to work with low utilization. To motivate the work presented here, we use the

2474-0756/18/$31.00 ©2018 IEEE
DOI 10.1109/ICAC.2018.00024



(a) $Client = 300, \texttt{cpulimit} = 30\%$ (b) $Client = 300, \texttt{cpulimit} = 70\%$

(c) $Client = 700, \texttt{cpulimit} = 30\%$ (d) $Client = 700, \texttt{cpulimit} = 70\%$

Fig. 1: 99.99th percentile latency (*t2.micro*) for *browsing*: different `cpulimit` with different arrival intensities.

typical SPEC TPC-W benchmark [2] to show that if the CPU credit consumption is controlled by the user, it can achieve better user-perceived performance by effectively expanding the time period that the applications enjoys high CPU bursts.

Figures 1 illustrate a first proof of concept of the effect of different CPU limits for TPC-W's browsing mix running on the t2.micro with 300 and 700 clients, respectively. The figures illustrate the normalized client latency when the default CPU scheduling is used (i.e., no `cpulimit`, dabbed as Amazon), when a certain `cpulimit` is used, and the target SLO (flat line). Two cases of `cpulimit` are considered: 30% and 70%. Assuming an especially challenging SLO latency (we require the 99.99% percentile of latencies to be below a certain value), we see that different `cpulimit` values result in different violations but most importantly in different lengths of the time that the system operates with burstable performance under the initial credit deletion period. When `cpulimit` 30% is used, the credit depletion period triples, see Figure 1(a), for `cpulimit` 70% it nearly doubles (see Figure 1(b)). The moment credit depletion completes, latencies increase dramatically. The credit depletion period is not as pronounced if the workload is heavier, see Figures 1(c)-1(d) with 700 clients. The ideal `cpulimit` is a function of the workload intensity (expressed by the number of clients), workload type (for this experiment we used the browsing mix, results are different for the ordering or shopping mix), and the desired SLO level.

The purpose of this paper is to devise an autonomic scheduling framework to adjust `cpulimit` on-the-fly in order to deal effectively with dynamic workloads (i.e., changing client intensities, changing workload types, and even changing SLOs) so that we can maximize the efficiency of the burstable capacity/spare resource while meeting SLOs. The challenge lies in the huge search space of `cpulimit` and expensive

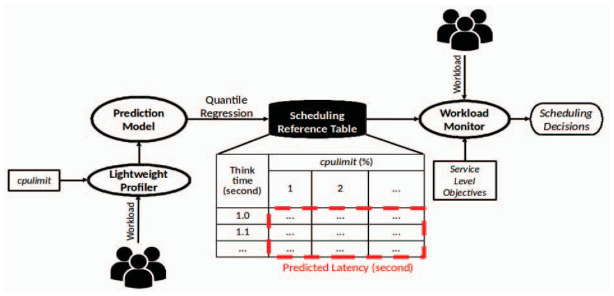


Fig. 2: Overview of CEDULE.

profiling. To this end, we combine light-weight profiling with an analytical model driven by recent research in quantile regression [3] to determine the optimal `cpulimit` that maximizes the efficiency of initial credits. We also incorporate the migration cost (when credit is depleted) in our analytical model to reflect the state-of-the-practice. We illustrate the effectiveness of the proposed scheduling framework using single-tier and multi-tier applications, under both static and dynamic workload conditions.

The remainder of this paper is organized as follows. Section II describes each step of the adaptive scheduling framework, Section III evaluates the proposed scheduling framework on Amazon EC2. In Section IV previous work is discussed and Section V concludes the paper.

II. Methodology

In this section, we present the proposed Credit Efficiency-aware scheDULing framEwork (CEDULE). CEDULE integrates empirical measurements (lightweight profiling) with an analytical method (quantile regression) in a synergistic manner, to find the best `cpulimit` for a given load (number of requests per second) that can maximize credit efficiency while meeting a predefined SLO.

A. Overview

The goal of CEDULE is to achieve a win-win scheduling solution for both the cloud service provider and its end users by enabling smart sharing of resources without violating user SLOs. From the user perspective, the benefit is cost reduction, otherwise it may be necessary to use a more expensive instance to meet SLO for sustained time periods. From the cloud provider perspective, if a user can achieve its SLO with less resources, spare resources can be multiplexed across more users. Previous work proposes migration once credit is depleted [24] but does not provide any solution for determining the ideal credit consumption rate to meet user performance requirements in order to reduce the frequency of migration. Frequent migration can have high monetary cost as two instances may run during the migration period and is also discouraged by service providers, e.g., Amazon has limited to 100 new instances that can start every day [11]. Figure 2 gives an overview of CEDULE, which is composed of three main components: profiler, prediction model, and scheduler, which are introduced in the following sections.

B. Lightweight Profiling

The straight-forward way to maximize the efficiency of spare resources is through exhaustive profiling. This can be costly and time consuming. Assuming L different CPU limit levels and A different loads, we need to conduct $L \times A$ profiling experiments. In addition, measuring the percentile latency requires a long time span to collect enough samples, especially for high percentiles. If t is the average time to achieve statistical stability in profiling, then the total cost of exhaustive profiling is $L \times A \times t$. CEDULE conducts lightweight profiling through sparse sampling and utilizes an analytical model to fill the rest of the search space. Assuming that the collected percentage samples from L is α and that the percentage of samples from A is β, then the total profiling cost becomes $\alpha \times \beta \times L \times A \times t$, which is $\alpha \times \beta$ times smaller than the exhaustive one. We feed the profiling results to our prediction model for estimating the tail latency. The prediction model is introduced in the next subsection.

C. Prediction Methodology

We propose a prediction methodology which takes the profiling data, SLO, and monitored system load as inputs and computes the lowest `cpulimit` as output so that the SLO is met and the efficiency of spare resources is maximized. The core of this prediction methodology is an analytical model based on quantile regression.

1) Problem Formulation: We define the computation of the ideal `cpulimit` as an optimization problem. The spare resources are measured by credits. We define Credit Efficiency (CE) as the average credit depletion time (T_d) minus the average migration time (T_m) under a given SLO constraint. Therefore, the problem of maximizing the efficiency of spare resources is equivalent to maximizing Credit Efficiency as follows:

$$\begin{aligned} \text{maximize} \quad & \mathrm{T_d - T_m} \\ \text{subject to} \quad & P_i \le \text{SLO}, \end{aligned} \tag{1}$$

where P_i is the i-th percentile of latency and its value depends on CPU throttling, i.e., $P_i = f(cpulimit)$. The migration time T_m depends on the migration approach, e.g., T_m can even be 0 if live migration is used, if proactive migration [24] is used, then the migration time equals to the time of copying the application status from the old instance to the new instance. The credit depletion time T_d depends on the initial credit C_i, credit earning rate R_e, credit consumption rate R_c (which is usually a function of $cpulimit$: $R_c = f(cpulimit)$), and system utilization ρ, where R_e and R_c are defined by the service provider. Credit is only consumed when the system is busy. Two different mechanisms affect credit earning: one is the default fixed credit earning rate, and the other is only earning credit when the system is idle. We add a parameter k to capture the credit earning mechanism: $k = \frac{1}{1-\rho}$ when the credit earning rate is fixed and $k = 1$ when credit earning occurs only during the time that the system is idle. T_d can be computed using the following equation:

$$C_i + T_d \times (R_e \times k \times (1-\rho) - R_c \times \rho) = 0. \tag{2}$$

Since system utilization depends on the arrival rate λ and instance service rate μ (when there is no CPU throttling), based on the Utilization Law [1], $\rho = \frac{\lambda}{\mu}$. For burstable instances, the service rate also depends on throttling if $cpulimit$ is used, therefore $\rho = \frac{\lambda}{\mu \times cpulimit}$. T_d can be computed as:

$$T_d = \frac{C_i}{R_c \times \frac{\lambda}{\mu \times cpulimit} - R_e \times k \times (1 - \frac{\lambda}{\mu \times cpulimit})}. \tag{3}$$

To summarize, the inputs of the model are:

- Load λ in terms of mean arrival rate: here we assume exponential inter-arrival times as it represents the typical online service behavior [4], note that the load can change over time to reflect the dynamic nature of the workload.
- Burstable service rate μ, i.e., when no throttling is applied.
- Initial credit C_i, credit earning rate R_e, and credit consumption rate R_c, as defined by the cloud service provider.
- Profiling results: sparse samples of the percentile latency under the target `cpulimit`.
- Service Level Objective (SLO).

The output of the model is the optimal $cpulimit$ with maximized Credit Efficiency without violating the user SLO.

The above problem formulation clearly shows how the `cpulimit` can be used as a control knob to adjust Credit Efficiency. However, its impact on Credit Efficiency is not straightforward and depends on several factors. In practice, a too low `cpulimit` may result in a too large P_i and thus violate the SLO, a high `cpulimit` may cause low Credit Efficiency and could fail to achieve the scheduling objective. Therefore, we need to model the impact of `cpulimit` on tail latency and Credit Efficiency. Next we introduce the analytical model, which is based on quantile regression.

2) Analytical Model: In order to identify the optimal `cpulimit`, we need to predict the percentile latency P_i for a given `cpulimit` and load. Here we build our analytical model upon quantile regression [3]. Quantile regression is a statistical inference method used for estimation and extrapolating the relationship between conditional quantile functions. The regression model for quantile level τ is given by

$$Q_\tau(y_i) = \beta_0(\tau) + \beta_1(\tau)x_{i1} + \ldots.. + \beta_p(\tau)x_{ip}, \quad i = 1, \ldots., n \tag{4}$$

where $\beta_j(\tau)s$ are estimated by solving the minimization problem:

$$\underset{\beta_0(\tau),\ldots,\beta_p(\tau)}{\text{minimize}} \sum_{i=1}^{n} P_\tau \left(y_i - \beta_0(\tau) - \sum_{j=1}^{p} x_{ij}\beta_j(\tau) \right) \tag{5}$$

where $P_\tau(r) = \tau max(r,0) + (1-\tau)max(-r,0)$. The function $P_\tau(r)$ is referred to as the check loss, because its shape resembles a check mark. For each quantile level τ, the solution to the minimization problem yields a distinct set of regression coefficients.

Quantile regression takes a number of samples as inputs and outputs the estimated coefficients β_i, which are calculated to minimize the prediction error on a particular quantile τ. Each input sample includes a set of independent variables x

and a response variable y. Apart form the individual independent variables it also takes into consideration the relationship among all independent variables. The error of the loss function is minimized via numerical optimization. The τ-th quantile loss function assigns a weight τ to underestimate errors and $(1-\tau)$ to overestimate ones. The main advantage of quantile regression is that it does not require any assumptions regarding the underlining distribution of the data [18] and is robust to non-normal errors and outliers compared to linear regression models.

Quantile regression only admits a one-dimensional sample of data as input. Our latency prediction model needs to consider both `cpulimit` and load, which form a two-dimensional space. Therefore, we extend the original quantile regression in [3] by first training the model using a fixed load while varying `cpulimit` to collect latency samples in the profiling step. Here, the `cpulimit` values are independent variables and the latency samples for each `cpulimit` are input samples. As a second step, we fix `cpulimt` and vary the loads to train a second model. The loads now become the independent variables and the latency samples are used as input samples. We integrate the models trained in these two steps into a complete model. Since the first model and the second model can predict the same case (i.e., for a given `cpulimit` and load), we calibrate the results by averaging the prediction results from each model[1]. Finally, we use the complete model to populate the Scheduling Reference Table in Figure 2.

D. Scheduler

The scheduler takes the load as input and searches the optimal `cpulimit` in the scheduling reference table to maximize Credit Efficiency while meeting the user SLO. More specifically, the scheduler chooses the smallest `cpulimit` with percentile latency smaller than or equal to the given SLO. This is because based on Eq 3, the smallest CPU limit has the longest depletion period and thus maximizes Credit Efficiency. When the workload is dynamic, `cpulimit` needs to be adjusted based on the monitored load to avoid SLO violations. We implement a fixed observation window of (e.g., 5 seconds) to measure the load of the incoming requests and adapt `cpulimit` based on the scheduling reference table.

III. Experimental Evaluation

In this section we evaluate CEDULE via experimentation on the Amazon EC2. First, we study the prediction error of the proposed methodology. Then, we evaluate the SLO violations. We compare the performance of our methodology against the default mechanism of t2.micro, in terms of SLO violations and the period of CPU credit depletion. Finally, we briefly compare the credit depletion period calculated by the analytic model with the one observed during the experiments.

[1]A more sophisticated way of calibration can be used here, but this is out of the main scope of this paper.

A. Experimental Setup

1) System overview: We evaluate CEDULE using single-tier and multi-tier benchmarks on Amazon EC2. For our experiments, we use t2.micro instances with Ubuntu Server 16.04 LTS. Each t2.micro instance has one vCPU, 1 GB memory and 30 initial credits; other parameters (and more information) are given in Table I. Due to AWS limits on the maximum number of instances allowed in each region (i.e., no more than 10 or 20 depending on the availability zone), experiments were run in the Virginia and Ohio regions. In the following, the benchmarks used are briefly described.

Single-Tier Applications. We consider two single-tier benchmarks: Sysbench [7] and YCSB [6]. *Sysbench* is a CPU intensive application whose requests perform prime number calculation. In our experiments each job generates 50 requests (i.e., prime number calculations), and it is completed only when all requests have been executed. *YCSB* is used to generate requests to a `memcached` system (i.e., a distributed memory object caching system). In contrast to Sysbench, memory is an important resource for this benchmark, but CPU is still the dominant resource. Independently of the benchmark used, we consider a closed system with 100 users, with average think times between 1 and 12 seconds, to best approximate a user-driven, latency-aware workload.

Multi-Tier Applications. We use the multi-tier benchmark TPC-W [2] in our evaluation. TPC-W is a three-tier web application with a client server, front-end web server, and database server. Here, we use one t2.micro instance for each tier. Since the CPU utilization of the client server is always below 10 percent, there is no need to experiment with `cpulimit` on the client server. We apply the same `cpulimit` on the web server and the database server, striving for a balanced system during profiling and in the evaluation experiments. The number of clients during all experiments is kept constant at 300. The browsing mix of TPC-W is used. In order to generate the load of varying arrival intensities, the think time of the clients is varied between 0.1 and 2 seconds.

2) Workload: We evaluate how `cpulimit` adapts to static and dynamic workloads within a closed-system setting, consistent with the TPC-W specifications. System load is controlled by the user think time [2], therefore we control the average arrival rate of requests by varying the user think time. In the rest of this paper, we interchangeably use the terms *think time* and *arrival rate*.

Static workload: The arrival rate of the workload is kept static during the entire period of the experiment – this is a basic experiment and CEDULE's target is to identify the smallest `cpulimit` such that the SLO is met.

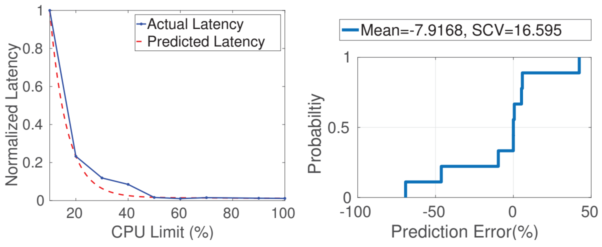
Dynamic workload: In this experiment, a fluctuating workload arrival rate is realized by changing the user average think time over time. This type of experiment allows to study the accuracy of prediction and the ability of the system to continuously adjust the optimal `cpulimit` under variable workload conditions.

B. CEDULE with Static Workloads

We first evaluate the accuracy of the latency prediction model by comparing model-predicted values to measured ones. To this end, we profile the single-tier and multi-tier benchmarks with latency samples for five different values of `cpulimit` (i.e., 10%, 20%, 50%, 70% and 100%). All other `cpulimit` values are extrapolated from the above experiments. Finally, note that `cpulimit`=100% is essentially the typical CPU usage, i.e., there is no user-mandated CPU throttling.

To evaluate CEDULE's robustness under stringent workload conditions, we use as SLO targets the 99th and 99.99th latency percentiles for the single-tier and multi-tier benchmarks, respectively. Model prediction results are compared against different `cpulimit` values and think times. For reasons of presentation clarity, we opt for normalized latencies rather than raw values in all results presented in this section.

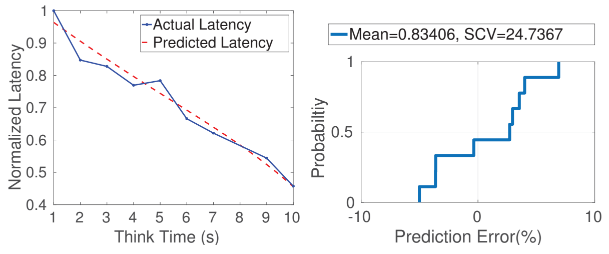
For the experiment presented in Figure 3, we set the think time to 12 seconds and vary `cpulimit`. After training the model, its accuracy is evaluated against actual experiments. The cumulative distribution function of the raw error is given in Figure 3(b). Here, negative values represent under estimation of the prediction model, while positive values correspond an over estimation, with an absolute average error less than 8%. As this experiment is executed for two hours, we evaluate the model robustness across four subsequent 30-minute windows. Across all windows results are consistent with those presented in Figures 3 and 3(b) and are not reported here due to lack of space.



(a) *Latency Comparison* (b) *Over/Under estimation CDF*

Fig. 3: Single-tier Sysbench prediction and error (t2.micro). Actual vs predicted 99th percentile, for $\texttt{cpulimit} = 10\%$ to $\texttt{cpulimit} = 100\%$ with an increment of 10%, and for think time set to 12 seconds. Note that model training is done with only a subset of `cpulimit` values , i.e., 10%, 20%, 50%, 70% and 100%.

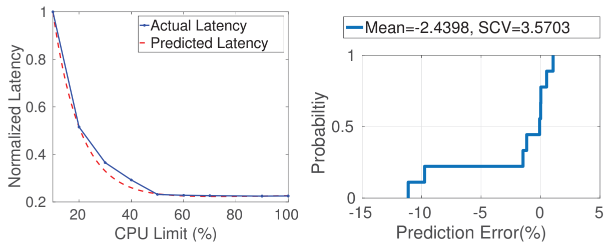
Next, we evaluate the predicted and measured 99th percentile latency for different load intensities, as expressed by different user think values, see Figure 4. fo this experiment, we set `cpulimit` to 30%. Consistent with the previous experiment, we use a sparse number of think times for model training (i.e., 1s, 2s, 5s, 7s and 10s) and extrapolate others from the model. Experimental and prediction results are in excellent agreement with average errors ever lower than 1% (see Figure 4(b)).



(a) *Latency Comparison* (b) *Over/Under estimation CDF*

Fig. 4: Single-tier Sysbench prediction and error (t2.micro). Actual vs predicted 99th percentile, for think time $Z = 1$s to $Z = 10$s with an increment of 1s, and for $\texttt{cpulimit} = 30\%$. Model training is done with a subset of think values, i.e., 1s, 2s, 5s, 7s and 10s.

Similar experiments are also done for the YCSB benchmark. For the first experiment, the think time is 5 seconds and `cpulimit` is varied. Results are shown in Figure 5, showing average error not larger than 2.5%. Similar to Sysbench, we also present results that show how the prediction model adapts to varying think times, see Figure 6. As in the the Sysbench base, average prediction errors are minimal.

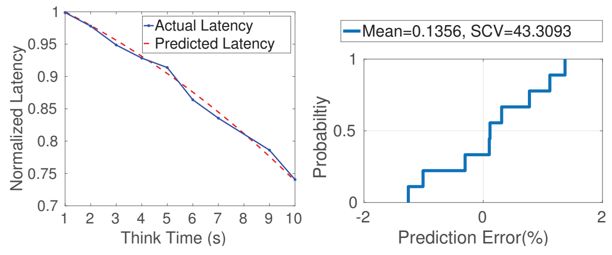


(a) *Latency Comparison* (b) *Over/Under estimation CDF*

Fig. 5: Single-tier YCSB (memcached system) prediction and error (*t2.micro*). Actual vs predicted *99th* percentile, for $\texttt{cpulimit} = 10\%$ to $\texttt{cpulimit} = 100\%$ with an increment of 10%, and for think time set to 5 seconds. Model training is done with only a subset of `cpulimit` values , i.e., 10%, 20%, 50%, 70% and 100%

.

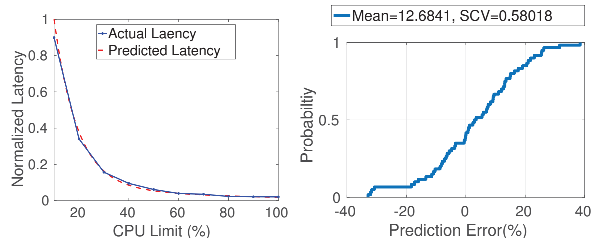
For the multi-tiered workload (TPC-W), we collect data via light-weight profiling for experiments with 300 customers and average think time equal to 1 second. For the more challenging multi-tiered case, we set the SLO percentile latency to 99.99 for a limited number of `cpulimit` values , i.e., 10%, 20%, 50%, 70% and 100%. In order to test the accuracy of prediction we also run 10 experiments for different CPU limits (10% to 100% with an increment of 10%), all other parameters are the same. Similarly to the single-tier experiments, we evaluate the model effectiveness for 6 consecutive windows in time – the model prediction is consistently robust. Figure 7 shows results across all time windows. The figure illustrates actual and predicted latencies during the entire duration of the experiment (one hour) and the CDF of prediction errors for



(a) *Latency Comparison* (b) *Over/Under estimation CDF*

Fig. 6: Single-tier YCSB (Memcached) prediction and error (t2.micro). Actual vs predicted *99th* percentile, for think time $Z = 1$s to $Z = 10$s with an increment of 1s, and for $\texttt{cpulimit} = 70\%$. Model training is done with a subset of think values, i.e., 1s, 2s, 5s, 7s and 10s.

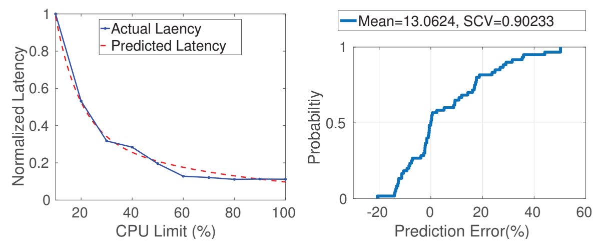
the web server. We observe that the predicted latency closely follows the same trends as the actual one. The point where latency errors are large is for `cpulimit` equal to 10%. This is an outcome of the low utilization and the fact that we do not sample enough data, especially for the stringent 99.99 percentile. Despite the complex interaction patterns between the database server and the web server in TPC-W and the very stringent 99.99 percentile target latency, the prediction model is quite robust, with mean latency error close to 12%, see Figure 7(b). Figure 8 presents similar results for the database server and further confirms the robustness of prediction.



(a) *Latency Comparison* (b) *Over/Under estimation CDF*

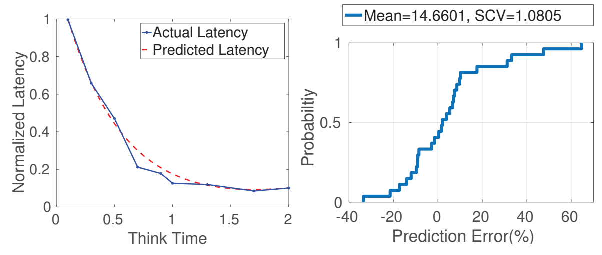
Fig. 7: TPC-W web server prediction and error (t2.micro). Actual vs predicted $99.99th$ percentile, for $\texttt{cpulimit} = 10\%$ to $\texttt{cpulimit} = 100\%$ with an increment of 10%, and for think time set to 1 second.

Similarly to the single-tier experiments, we test model accuracy for TPC-W for varying arrival rates. For this set of experiments, `cpulimit` is fixed to 70 percent on both web and database servers. The value of think time is varied between 0.1 to 2 seconds. The varying arrival rate results in different web and database server utilization levels. Profiling is done with 5 think time values (i.e., 0.1s, 0.5s, 0.1s, 1.5s, and 2s). The reason for selecting think time values between 0.1 second to 2 seconds is that, beyond these values the system utilization drops below 70 percent and `cpulimt` cannot truly affect the system. To verify model robustness, we also do experiments for 10 thinking times . The comparison of latency and CDF



(a) *Latency Comparison* (b) *Over/Under estimation CDF*

Fig. 8: TPC-W database server prediction and error (t2.micro). Actual vs predicted $99.99th$ percentile, for $\texttt{cpulimit} = 10\%$ to $\texttt{cpulimit} = 100\%$ with an increment of 10%, and for think time set to 1 second.

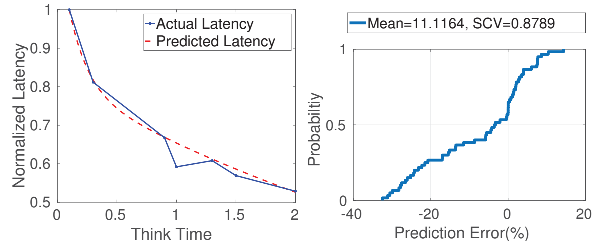


(a) *Latency Comparison* (b) *Over/Under estimation CDF*

Fig. 9: TPC-W web server(*t2.micro*) $99.99th$ percentile of actual and predicted latency and CDF of model errors for fixed $cpulimit = 70\%$ with varying arrival rates (x-axis).

of error percentage for web sever and database server are shown in Figures 9 and 10, respectively. For the web server we observe a higher mean error compare to other cases. That may be caused by interference of other VMs co-located on the same physical machine that hosts the web server instance [8, 9, 21].

A similar abnormal behavior is also observed for think time equal to 1 second in the database server, see Figure 10(a). The CDF of error for the database server is given in Figure 10(b). In general, errors are consistent with those reported for varying `cpulimit`.



(a) *Latency Comparison* (b) *Over/Under estimation CDF*

Fig. 10: TPC-W database server (*t2.micro*) $99.99th$ percentile of actual vs predicted latency and CDF of model errors for fixed $cpulimit = 70\%$ with varying arrival rates (x-axis).

C. CEDULE with Dynamic Workloads

In this section we study the accuracy of CEDULE when single-tier or multi-tier systems serve dynamic workloads. For this purpose, we first train our model as described in Section II, sampling the system latency for few configurations (i.e., different user think times and `cpulimit` values), and use quantile regression to derive the `cpulimit` for all the other possible configurations. Then, the effectuviness of CEDULE under a dynamically changing workload is tested for Sysbench and TPC-W. The dynamic workload is generated by varying the average think time for each client during the time that the experiment takes place. The observation window, i.e., the period of time during which the inter-arrival rate is observed by CEDULE to select the best `cpulimit` value, is just 1 minute long. The measured inter-arrival rate and the SLO value are input parameters to the model to calculate the smallest `cpulimit` that can meet the advertised SLO. If a change in the inter-arrival rate is observed during the observation window, a new `cpulimit` value is imposed on the system. In this section we present results for a fixed SLO value. We note that CEDULE is also robust if SLO is changed dynamically – such results are not presented here due to lack of space.

The results for the single-tier application are shown in Fig. 11. The experiment lasts 32 minutes during which $13,250$ requests are served. The user think time varies five times during the experiment and its value is between 5.6 and 9.1 seconds. The SLO (on the $99th$ percentile latency) is set to 10 seconds for the duration of the experiment. For the sake of presentation clarity, all values are normalized over the SLO. The figure shows that `cpulimit` varies as a function of the user think times, but with a small lag. This effect is due to the observation window because CEDULE observes the workload before changing to a new `cpulimit`. Note that longer user think times correspond to a smaller inter-arrival rate. Thus, `cpulimit` decreases when think time increases. Overall, the number of violations observed during the experiment is very small (i.e., less than 0.2%) and CEDULE successfully adapts `cpulimit` to the changing workload.

To generate the dynamic workload for TPC-W, think times are varied between 0.2 to 1.8 seconds. In this experiment we started a new t2.micro instance with 30 credits. Each experiment ran until the amount of available credits is depleted. Figure 12 shows the change in the `cpulimit` overtime, the figure also reports the arrival rate in the web server, the $99.99th$ percentile latency, and the SLO. The spikes in the arrival rate to the web server are due to the observation windows during which the incoming workload is changing but the new `cpulimit` is not yet activated. The accumulated $99.99th$ percentile latency value is very close to the SLO value, this clearly demonstrates that the proposed solution chooses `cpulimit` very efficiently. Overall, we only observe 0.0086 percent violations during the entire period.

D. Benefits over the Default Amazon Mechanism

In order to fully understand the efficiency of our adaptive strategy, we compared the credit depletion period (i.e., the time before an instance runs out of credits) of two systems executing the benchmarks: in the first case, the system is adopting the methodology presented in this paper, thus trying to save resources while complying with the SLO, in the second case the system is using the default Amazon scheduling. This experiment is also done for the single-tier and multi-tier systems, using Sysbench and TPC-W benchmarks, respectively.

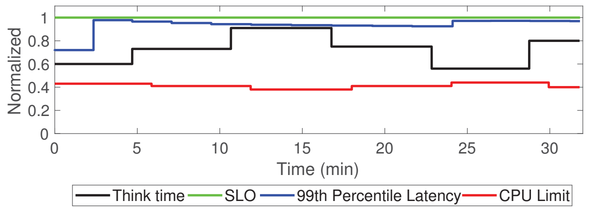


Fig. 11: Single-tier Sysbench: $99th$ percentile latency, `cpulimit`, users' think time and SLO for dynamic workload. All the values are normalized over the SLO. Note that, the inter-arrival rate decreases as the think time increases.

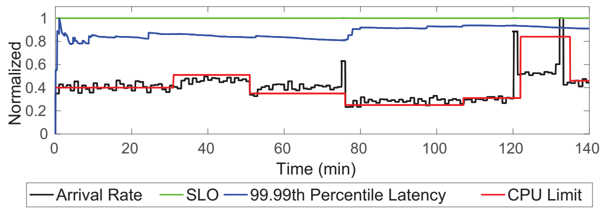


Fig. 12: Multi-tier TPC-W: web server $99.99th$ percentile latency, `cpulimit`, request arrival rate (jobs/minute) and SLO for dynamic workload. All the values are normalized over the SLO.

For the single-tier case, we execute the Sysbench benchmark (dynamic workload) with CEDULE and with the default Amazon scheduling. Both experiments start with 33 credits and the $99th$ percentile latency of completing requests is observed for 40 minutes. Figure 13 depicts the $99th$ percentile latency of both experiments as a function of time. The SLO (i.e., 10 seconds) is also depicted in the figure. For the sake of clarity, all values are normalized over the highest observed value and y-axis is in logarithmic scale. As expected, the CEDULE makes credits last longer by making better use of spare resource while the Amazon scheduling essentially wastes these space resources as credits are depleted faster. Once depletion happens, the $99th$ percentile latency grows and the SLO is violated. It is also interesting to note that the Amazon scheduling is exhausting all its credits before 30 minutes of work. In fact, we have observed that t2.micro instances start throttling the CPU before the number of available credits reaches 0. This happens when the amount of credits of an instance is between 6 and 7.

We also present here a similar experiment for TPC-W (dynamic workload). All tiers for both CEDULE and the

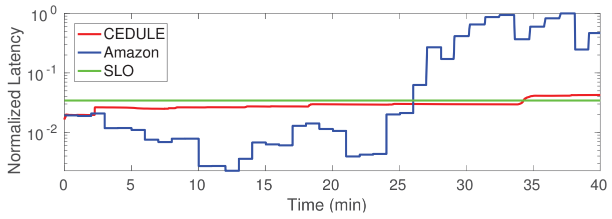


Fig. 13: Comparison of the normalized 99th percentile latencies (t2.micro) achieved by CEDULE and the default Amazon policy for Sysbench. CEDULE uses different a `cpulimit` for different arrival intensities.

default Amazon policy start with 36 credits. The experiment duration goes till the point when one of the tiers is fully throttled on both policies. The $99.99th$ latency percentile for the web server is given in Figure 14. Similar to the Sysbench experiment, the default Amazon scheduling depletes the available credits fast. CEDULE applies `cpulimit` on both the web and database servers and results in superior credit utilization. The accumulated $99.99th$ percentile latency that always closely follows the SLO, reflects the impressive accuracy of CEDULE's `cpulimit` calculation, nearly tripling the time to credit depletion.

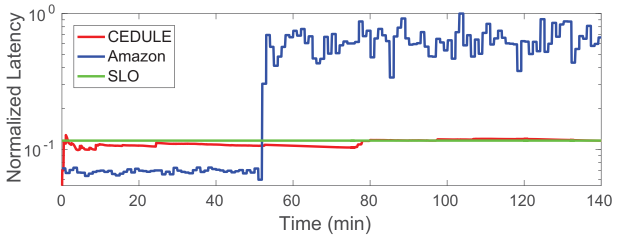


Fig. 14: Comparison of the normalized (log scale) $99.99th$ percentile latency (t2.micro) achieved by CEDULE and the default Amazon policy for TPC-W. CEDULE uses a different `cpulimit` for different arrival intensities.

E. Benefits over Exhaustive Profiling

As described in Section II, CEDULE uses light weight profiling to identify and adopt the best `cpulimit` to meet the SLO under different load levels. The experimental results show that CEDULE can accurately predict the latency for varying `cpulimit` and arrival rate values. Moreover, the cost of CEDULE is significantly lower than exhaustive profiling. For example, when considering a dynamic workload, CEDULE requires only 25 profiling experiments (i.e., 5 `cpulimit` $\times$ 5 users' think time) to create a reference table with 100×100 entries, while exhaustive profiling requires $10,000$ experiments. The time for profiling experiments and building the reference table for the dynamic workload case is shown in Table II. In order to accurately measure latency with the desired percentile, profiling experiments should capture enough samples. In fact, the profiling time of each experiment depends on the application behavior and workload. In particular, YCSB requests are those with the longest service time, whereas Sysbench and TPC-W generate shorter requests. For this reason, the time required to complete the profiling procedure for YCSB is longer than the time for Sysbench and TPC-W.

F. Comparison of Depletion Period: Analytic Model vs Experimental Results

The Depletion time of the CPU credits is modeled in equation 3. Given the value of CPU utilization and `cpulimit`, the credit depletion time is calculated so that the application can schedule a migration before running out of credits.

Equation 3 is checked against some of our experiments. In fact, it is possible to derive the credit depletion time of an instance by observing the amount of its available credits at the beginning of the experiment and its average CPU utilization while it serves the incoming requests. Results for the static and dynamic workload cases are shown in Table III, showing that the predicted credit deletion time is remarkably close to the real one.

We note that CEDULE minimizes the migration penalty by reducing the number of migrations. Indeed, credit utilization in CEDULE depends on the current workload and SLOs. This allows the instance to extend the CPU depletion period and results in a lower number of migrations compared to the default Amazon policy. For example, the credit depletion period for the default instance in Figure 14 is almost 55 minutes. Instead, by applying CEDULE, this period is extended to 145 minutes. Thus, considering a 24-hour period, 27 new instances must be launched if the system adopts the default Amazon EC2 strategy, instead only 10 can be launched with CEDULE.

Finally, to avoid long service interruptions, one must also account for migration time when analyzing the cost of this operation. Tools like CMT[2] – together with CEDULE allow us to proactively migrate the instance as soon as the number of available credits approaches zero. In this case, the time required for migration depends only on the amount of data to be transferred. For the experiments presented here, it varies from few tens of MB in the single-tier system to 400 MB on average for the multi-tier one.

IV. Related Work

Resource management in cloud computing has been deeply studied in the literature. Many different techniques and frameworks have been proposed [12, 17, 14, 15, 19, 20], focusing on the cloud users' Quality of Service (QoS) and SLOs. Some strategies account for interference among VMs hosted on the same physical machine that may degrade the performance of individual instances. Wang et al. [12] proposed a *Fuzzy Model Predictive Control* to automatically manage resources while complying with QoS and SLOs. Javadi et al. [19] introduce DIAL, an interference-aware load balancer that can reduce long tail latencies in cloud-deployed applications. Other

[2]https://github.com/marcosnils/cmt

TABLE II: Cost comparison between exhaustive and light profiling for different benchmarks.

Benchmark	Method	Ref. Table `cpulimit` entries	Ref. Table loads entries	# profiling exp.	Profiling time [min.]	Total profiling time [days]
Sysbench	Light Exhaustive	100	100	25 10000	9.27	0.16 64.35
YCSB	Light Exhaustive	100	100	25 10000	25.88	0.45 179.72
TPC-W	Light Exhaustive	100	100	25 10000	10	0.17 69.44

TABLE III: Estimated and observed credits depletion time for static and dynamic workloads.

Workload	# credits	U_{CPU} [%]	Estim. T_d [min]	Real T_d [min]
Static	23.5	100	25.56	26
Static	23.5	55	51.11	52
Dynamic	25	100	27.77	27
Dynamic	25	62	48.08	45

techniques try to predict the workload of cloud applications and allow the final users to scale their VMs based on prediction. For example, in [14] a Kalman-based estimator is adopted to predict the workload and resource allocation is performed through different algorithms. Frameworks to make resource management more effective are also proposed in [17], that investigated a *pack-centric* framework to group VMs according to resource sharing and collocation requirements. Liu et al. [15] described NetAnalytics, a monitoring system to study performance of cloud data centers and automate resource management by analyzing network data. Some frameworks are proposed to improve cost efficiency and resource utilization of cloud applications, e.g., iCSI [20] introduces a cloud garbage VM collector to detect inactive instances by collecting data from the VMs. Morris et. al. [16, 25] devise mechanisms to increase computational sprinting using DVFS to boost processor clock rates and AWS burstable instances. Their work is based on off-line system measurements and uses a machine learing model to predict the response time model under different sprinting strategies but they assume a priori knowledge of the workload running conditions (e.g., arrival rate).

Although On-Demand instances are relatively new, previous works have introduced models for their analysis and proposed different ways to get best advantages from their features. Some of these works take into consideration the previous generation Amazon EC2 burstable instances (i.e., T1 instances). For example, Wen et al. [13] statistically characterized the behavior of a t1.micro virtual machines (VMs), and proposed to limit instances CPU consumption by injecting delays to make VMs provide better performance with lower prices. Since 2014, Amazon Web Services (AWS) introduced T2 instances, the new generation instances that replaced the T1 ones. T2 instances have a much better performance profile, and the main difference with respect to T1 instances is the introduction of credits to manage burst of performance. Many authors [10, 23, 24] focus on investigating the T2 model and its performance variations due to the amount of available credits. Leitner and Scheuner [10] proposed a basic model to analytically and empirically study T2 instances. This work also analyzes the credit boosting idea to obtain better performance with a lower price. Unfortunately, this strategy has some limitations due to constraints on the number of time a T2 instance can be started or rebooted with full amount of credits (e.g., t2.micro instances can be launched 100 times in a 24-hour period) [11]. Wang et al. [23] investigated the mechanism used for On-Demand instances. For this purpose, they consider VMs from both Amazon EC2 and Google Cloud Engine. They note that, due to deterministic regulation mechanisms, network and CPU performance of burstable instances are subject to high dynamism. They also proposed some possible exploitations of burstable instances, such as *passive backup for spot instances* [22] and *temporal multiplexing of burstable instances*. [24] focuses on how to extend the lifetime of credits in single-tier and multi-tier systems by adopting `cpulimit`, surpassing the performance of the delay strategy proposed in [13] that can negatively affect the end-to-end execution time making it longer. Besides extending the credits lifetime and the performance of the system, the approach presented in [24] also allows users to reduce migration cost by decreasing the migration frequency.

To the best of our knowledge, there are not available techniques for bursting instances that can dynamically adapt to variations in the workload while making the credits deplete as late as possible. With bursting instances being a relative new topic, no works have considered in depth mechanisms to make their credits last longer. Starting from the results obtained and presented in [24], we introduce that is based on a novel autonomic strategy that combines `cpulimit` and quantile regression to make the system save credits while complying with user-defined strict SLOs.

V. Conclusions

This work presents , an autonomic scheduling framework, that can maximize the efficiency of spare resources while preserving SLOs in burstable cloud instances. The core component is a tail latency prediction model driven by quantile regression that can predict the tail latency under a given load and `cpulimit`. The symbiosis of lightweight profiling and analytical modeling significantly reduces the overhead of collecting heuristics for training and enables the proposed work to be useful in practice. Extensive experimental evaluation using both single-tier and multi-tier applications on Amazon EC2 verifies the prediction accuracy of the proposed tail latency prediction model (with prediction errors ranging from 1% to 15%) as well as the effectiveness of in terms of improving the efficiency of spare resources without violating SLOs.

Acknowledgements: This work is partially supported by NSF grants CCF-1218758, CCF-1649087, and CCF-1756013.

References

[1] Edward D Lazowska et al. *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., 1984.

[2] Wayne D Smith. *TPC-W: Benchmarking an ecommerce solution*. 2000.

[3] Roger Koenker and Kevin F Hallock. "Quantile regression". In: *Journal of economic perspectives* 15.4 (2001), pp. 143–156.

[4] Ningfang Mi et al. "Burstiness in multi-tier applications: Symptoms, causes, and new models". In: *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer. 2008, pp. 265–286.

[5] Rajkumar Buyya et al. "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility". In: *Future Generation computer systems* 25.6 (2009), pp. 599–616.

[6] Brian F Cooper et al. "Benchmarking cloud serving systems with YCSB". In: *Proceedings of the 1st ACM symposium on Cloud computing*. ACM. 2010, pp. 143–154.

[7] Alexey Kopytov. "SysBench manual". In: *MySQL AB* (2012).

[8] Dejan Novakovic et al. "Deepdive: Transparently identifying and managing performance interference in virtualized environments". In: *Proceedings of the 2013 USENIX Annual Technical Conference*. EPFL-CONF-185984. 2013.

[9] Yasaman Amannejad, Diwakar Krishnamurthy, and Behrouz Far. "Detecting performance interference in cloud-based web services". In: *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*. IEEE. 2015, pp. 423–431.

[10] Philipp Leitner and Joel Scheuner. "Bursting with Possibilities–An Empirical Study of Credit-Based Bursting Cloud Instance Types". In: *Utility and Cloud Computing (UCC), 2015 IEEE/ACM 8th International Conference on*. IEEE. 2015, pp. 227–236.

[11] Rohit K Mehta and John Chandy. "Leveraging checkpoint/restore to optimize utilization of cloud compute resources". In: *Local Computer Networks Conference Workshops (LCN Workshops), 2015 IEEE 40th*. IEEE. 2015, pp. 714–721.

[12] Lixi Wang et al. "Qos-driven cloud resource management through fuzzy model predictive control". In: *Autonomic Computing (ICAC), 2015 IEEE International Conference on*. IEEE. 2015, pp. 81–90.

[13] Jiawei Wen et al. "Less can be more: Micro-managing vms in amazon ec2". In: *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*. IEEE. 2015, pp. 317–324.

[14] Joseph Doyle et al. "Cloud Instance Management and Resource Prediction For Computation-as-a-Service Platforms". In: *Cloud Engineering (IC2E), 2016 IEEE International Conference on*. IEEE. 2016, pp. 89–98.

[15] Guyue Liu et al. "NetAlytics: Cloud-Scale Application Performance Monitoring with SDN and NFV". In: *Proceedings of the 17th International Middleware Conference*. ACM. 2016, p. 8.

[16] Nathaniel Morris et al. "Sprint Ability: How Well Does Your Software Exploit Bursts in Processing Capacity?" In: *2016 IEEE International Conference on Autonomic Computing, ICAC 2016, Wuerzburg, Germany, July 17-22, 2016*. 2016, pp. 173–178. DOI: 10.1109/ICAC.2016.61. URL: https://doi.org/10.1109/ICAC.2016.61.

[17] Yi Wang et al. "Demonstrating Scalability and Efficiency of Pack-Centric Resource Management for Cloud". In: *Cloud Computing (CLOUD), 2016 IEEE 9th International Conference on*. IEEE. 2016, pp. 849–854.

[18] Yunqi Zhang et al. "Treadmill: Attributing the source of tail latency through precise load testing and statistical inference". In: *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE. 2016, pp. 456–468.

[19] Seyyed Ahmad Javadi and Anshul Gandhi. "DIAL: Reducing Tail Latencies for Cloud Applications via Dynamic Interference-aware Load Balancing". In: *Autonomic Computing (ICAC), 2017 IEEE International Conference on*. IEEE. 2017, pp. 135–144.

[20] In Kee Kim et al. "iCSI: A Cloud Garbage VM Collector for Addressing Inactive VMs with Machine Learning". In: *Cloud Engineering (IC2E), 2017 IEEE International Conference on*. IEEE. 2017, pp. 17–28.

[21] Scott Votke, Seyyed Ahmad Javadi, and Anshul Gandhi. "Modeling and Analysis of Performance under Interference in the Cloud". In: *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE. 2017, pp. 232–243.

[22] Cheng Wang et al. "Exploiting Spot and Burstable Instances for Improving the Cost-efficacy of In-Memory Caches on the Public Cloud". In: *Proceedings of the Twelfth European Conference on Computer Systems*. ACM. 2017, pp. 620–634.

[23] Cheng Wang et al. "Using Burstable Instances in the Public Cloud: Why, When and How?" In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 1.1 (2017), p. 11.

[24] Feng Yan et al. "How to Supercharge the Amazon T2: Observations and Suggestions". In: *Cloud Computing (CLOUD), 2017 IEEE 10th International Conference on*. IEEE. 2017, pp. 278–285.

[25] Nathaniel Morris et al. "Model-driven computational sprinting". In: *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*. 2018, 38:1–38:13. DOI: 10.1145/3190508.3190543. URL: http://doi.acm.org/10.1145/3190508.3190543.

[26] *Amazon Elastic Compute Cloud*. https://aws.amazon.com/ec2/. Accessed: 2018-03-22.

[27] *Google Compute Engine*. https://cloud.google.com/compute. Accessed: 2018-03-22.

[28] A Marletta. *CPU usage limiter for Linux*. http://cpulimit.sourceforge.net/.

[29] *Microsoft Azure*. https://azure.microsoft.com/. Accessed: 2018-03-22.