IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems

BATS: Budget-Constrained Autoscaling for Cloud Performance Optimization

A. Hasan Mahmud
Florida International University
amahm008@cs.fiu.edu

Yuxiong He
Microsoft Research
yuxhe@microsoft.com

Shaolei Ren
University of California, Riverside
sren@ece.ucr.edu

***Abstract*—Autoscaling has become an integral feature of cloud computing services, allowing users to dynamically scale the cloud resources on demand for both performance and cost. Moreover, recent survey shows the importance of satisfying long-term budget constraints (e.g., monthly or yearly) for cloud users. However, meeting such constraints while optimizing delay performance is challenging: it requires the knowledge of complete offline information such as workload demand over the entire budgeting period, which is difficult to predict accurately. This paper proposes a new autoscaling system, BATS, which optimizes delay performance while meeting long-term budget constraints using only past and instantaneous workload information. Analytically, we prove that, for arbitrary workload arrival, the autoscaling algorithm of BATS achieves close-to-optimal performance even compared to the optimal solution that has complete offline information. Empirically, we build BATS autoscaler as a user-friendly service for running applications on Windows Azure. The experimental results show that BATS achieves both lower cost and less delay compared with the state-of-art threshold-based autoscaling solutions. We also run simulation studies to complement the implementation results, demonstrating the effectiveness, scalability and robustness of BATS for reducing both average and tail latency under various workload scenarios.**

I. Introduction

Elasticity and scalability are important features of the emerging cloud computing systems, where virtual machine (VM) instances are dynamically purchased/released using autoscaling techniques in an automated fashion. While autoscaling VM instances, cloud users seek two major benefits, i.e., good performance and low expenses. In particular, they often have a cost budget in mind and desire the best performance within their budget. Towards this end, we develop a novel autoscaling algorithm and a full-fledged system to optimize delay while satisfying user's long-term budget constraint (e.g., monthly or yearly budget). Our study focuses on delay-sensitive cloud applications, e.g., web services, for which application performance is measured by the delay of responses.

Supporting budget constraint is essential for common business practice: as shown in a recent survey [1] covering 1,000 data centers by Uptime Institute, over 80% data center operators/managers are given budgets by business departments or higher-level executives at the beginning of a budgeting period (e.g., typically, a month or a year). Such budget constraints are also commonly applied to universities and governments, which typically allocate annual IT operational budgets at the beginning of each fiscal year [2].

Meeting budget constraints while optimizing delay performance is challenging. Requesting more VMs at the current time will reduce the available budget for future uses, which may significantly degrade performance and/or exceed budget in the event of high future workloads. Hence, optimally scaling VM acquisitions requires complete offline information (e.g., workload demand) over the entire budgeting period, which is very difficult, if not impossible, to obtain in advance, especially considering highly dynamic workloads and possible traffic spikes due to breaking events [3]. Default autoscaling mechanisms offered by major cloud service providers, such as Amazon EC2 and Windows Azure, typically scale VM instances based on resource utilization indicators such as CPU and memory usage. For example, add a new VM instance or switch to a bigger VM instance when the current CPU utilization exceeds a certain user-specified threshold [4], [5]. Threshold based autoscaling cannot optimize the performance while satisfying the budget constraint because of the following limitations. A too low resource usage threshold may incur an unnecessarily high cost, while a too high threshold reduces the cost, but may result in an intolerable performance. It is difficult to decide optimal resource usage thresholds *a priori*, because their values depend on the user budget and the workloads, while the long-term future workload information is very difficult to accurately predict. Recent efforts on autoscaling for optimizing the performance under long-term budget constraints have primarily focused on evenly dividing budgets across time or predicting the long-term future workloads, neither of which applies to highly-dynamic delay-sensitive workloads in practice [6], [7], [8].

In this paper, in view of the practical difficulty in accurately predicting long-term future workloads, we develop an online autoscaling system, called BATS (Budget-constrained AuToScaling), that dynamically scales VM instances to optimize the delay performance while satisfying user budget constraint in the long run. The core of BATS is an online autoscaling algorithm we propose, which only requires the past and instantaneous workload to make effective scaling decisions. The key idea of our algorithm is to keep track of the budget deficit online and incorporate it into the online autoscaling decision: if the actual VM expenses far exceed the expected cost, BATS tries to reduce the budget deficit by requesting fewer or smaller VM instances subject to the user-specified delay requirements. Leveraging Lyapunov optimization technique, we prove that the BATS produces a close-to-optimal delay performance compared to the optimal algorithm with offline information, while satisfying the budget constraint.

As a system, we build a *fully-automated* BATS autoscaler service on Windows Azure. BATS autoscaler only requires user

1526-7539/15 $31.00 © 2015 IEEE
DOI 10.1109/MASCOTS.2015.43

inputs on the desired delay performance and budget of their applications. It manages the performance monitoring, resource planning, and scaling of user applications automatically. We also combine BATS algorithm with a reactive module that monitors runtime performance and handles workload burstiness. The modular design of BATS autoscaler makes it easily adaptable to other cloud platforms such as Amazon EC2.

We evaluate the performance of BATS by running RUBiS benchmark workloads [9] on Windows Azure. The experimental results show that BATS achieves up to 34% less delay compared to the algorithm that evenly divides users budget over all time slots. Compared to the threshold-based scaling rules that are widely used by major cloud service providers, BATS reduces user cost by 10% while achieving a better delay performance. Moreover, the performance of BATS is very close to that of the optimal offline algorithm that knows complete future information. We also conduct extensive simulation study in terms of both average delay and 95^{th} percentile delay, showing the effectiveness of our algorithms on different performance metrics and its scalability on managing applications with hundreds of VMs. We show that BATS is truly autonomous: it does not need users to select appropriate algorithm parameters. BATS decides its parameters through online learning and adaptation.

In summary, the main contribution of our work is a *full-fledged* autoscaling solution, combining a provably-efficient algorithm and an implementation of the autoscaling service, to optimize delay performance while meeting users' long-term budget constraints that widely exist in practice.

II. Model and Problem Formulation

A. Model

We consider a discrete-time model by evenly dividing the budgeting period (e.g., typically a month or a year) into K time slots indexed by $t = 0, 1, \cdots, K-1$, each of which has a duration that matches the pricing policy of cloud service providers (CSPs). For example, each time slot can correspond to one hour if we subscribe to Windows Azure or Amazon EC2, both of which charge users for VM instances on an hourly basis.

• **Autoscaler:** We consider J types of VM instances (e.g., small, medium, large) where each VM instance is specified by a set of N resource configuration parameters $\mathscr{A}_j = \{a_{j,1}, \cdots, a_{j,N}\}$. Each parameter represents the provisioning of one resource type. For example, each small-type VM instance has one CPU core and 1.75GB RAM, while each medium one has two CPU cores and 3.5GB RAM on Azure [10]. An autoscaler scales VM instances over time. At time t, the autoscaler requests $m_j(t)$ type-j VM instances, whose price is $p_j(t)$. For notational convenience, we use the vectorial expression $\mathbf{m}(t) = [m_1(t), m_2(t), \cdots, m_J(t)]$ wherever appropriate. Given the autoscaling decision $\mathbf{m}(t)$, the cost incurred by the user at time t is expressed as

$$c(t) = \sum_{j=1}^{J} \left[p_j(t) \cdot m_j(t) \right]. \tag{1}$$

• **Workload:** Our study focuses on web services and hence, workloads are web requests. We denote by $\lambda(t) \in [0, \lambda_{\max}]$ the workload arrival rate at time t, where $\lambda_{\max}$ is the maximum possible arrival rate. Arrival rate can be measured by different metrics, such as the number of clients, VM CPU/memory utilization. To quantify the delay performance of web services, our work supports a variety of well-known metrics, such as average delay and tail delay (e.g., 95th-percentile latency). Without restricting our model to any particular performance metric, we use the general notion of $g(\lambda(t), \mathbf{m}(t))$ to represent the delay performance of interest during time t, which is jointly determined by the workload arrival rate $\lambda(t)$ and VM acquisition decisions $\mathbf{m}(t)$.

B. Problem Formulation

As workload arrival rate varies over time, the delay performance during high workloads should naturally be given a higher weight than that during low workloads when we evaluate autoscaling algorithms. Hence, our objective is to minimize the long-term delay performance over the entire budgeting period (i.e., K time slots), expressed as

$$\bar{g} = \sum_{t=0}^{K-1} \frac{\lambda(t) \cdot g(\lambda(t), \mathbf{m}(t))}{\sum_{t=0}^{K-1} \lambda(t)}, \tag{2}$$

where $g(\lambda(t), \mathbf{m}(t))$ is the delay performance at time t given the workload arrival rate $\lambda(t)$ and the scaling decision $\mathbf{m}(t)$. The term $\lambda(t) / \sum_{t=0}^{K-1} \lambda(t)$ is a weight that scales the delay at time t in proportion to the workload arrival rate. At time t, we set the maximum and minimum delay constraints denoted by

$$d_{\min} \leq g(\lambda(t), \mathbf{m}(t)) \leq d_{\max}, \ \forall t, \tag{3}$$

where the maximum delay constraint specifies the worst delay that can be tolerated subject to service level agreement (SLA), while the minimum delay threshold indicates that user experience improvement is negligible by letting the delay go below the threshold [11] and hence no need to over-request VM instances.

There may be a limit on the purchased resources specified by the CSP. For example, by default, a maximum of 20 virtual CPUs may be purchased from Azure Cloud Service unless approved for more [5]. We express such constraints as

$$\sum_{j=1}^{J} a_{j,n} \cdot [m_j(t)] \leq \bar{A}_n, \ \ \forall t \text{ and } \forall n = 1, \cdots, N, \tag{4}$$

where $a_{j,n}$ is the provisioned resource n associated with each type-j VM instance and $\bar{A}_n$ is the limit on resource n. Total cost needs to satisfy a long-term budget constraint

$$\sum_{t=0}^{K-1} c(t) \leq B, \tag{5}$$

where $c(t)$ is the incurred cost given by Equation (1). Note that the budget does not include bandwidth charge, which only depends on the workloads and cannot be *autoscaled*.

Note that we can rewrite the delay in Eqn. (2) as $\bar{g} = \frac{1}{K} \sum_{t=0}^{K-1} \lambda(t) \cdot g(\lambda(t), \mathbf{m}(t)) \cdot \frac{K}{\sum_{t=0}^{K-1} \lambda(t)}$, where, given workloads and budgeting period, the term $\frac{K}{\sum_{t=0}^{K-1} \lambda(t)}$ is constant. Hence, we can omit it in the following analysis for notational convenience, and present the (offline) problem formulation for

delay minimization as follows:

$$\mathbf{P1}: \quad \min_{\mathscr{M}} \frac{1}{K}\sum_{t=0}^{K-1} \lambda(t)\cdot g(\lambda(t),\mathbf{m}(t)) \tag{6}$$

$$s.t., \quad \text{constraints } (4),(5),(3), \tag{7}$$

where $\mathscr{M}$ denotes a sequence of scaling decisions over the budgeting period, which we need to optimize.

III. Algorithm Design and Analysis

This section presents our autoscaling algorithm, BATS, and analyze its performance. We first describe the inputs of BATS and then show how to make autoscaling decisions by using the information readily available at the current time step without requiring hardly-accurate long-term prediction. We formally prove that, for any workload, the performance of BATS is close to the offline optimal that has complete future information.

A. Obtaining Inputs to BATS

BATS requires two types of inputs:

• **Workload arrival rate** $\lambda(\mathbf{t})$: In practice, the workload predictor can estimate the instantaneous load, the arrival rate $\lambda(t)$, prior to the beginning of time t using some well-studied learning techniques (e.g., auto-regression analysis) [12]. Note that this prediction is short-term, only for the immediate next time slot, which is different from the long-term prediction of the entire budgeting period required by an offline algorithm. Many prior studies show that such instantaneous workload can often be predicated with a high accuracy [12]. Furthermore, we discuss how to handle unpredictable workload spikes in Section IV-B and quantify the impact of inaccurate prediction in Section VI-D.

• **Delay performance**: In general, the delay increases with increase on arrival rate, decrease on the number or size of VM instances. Nonetheless, the delay is also affected by a variety of other factors, such as queuing discipline and load balancing decisions (which may not always be controllable from users' perspective). Thus, it is challenging, if not impossible, to mathematically express the delay $g(\lambda(t),\mathbf{m}(t))$ as an explicit function of $\lambda(t)$, and $\mathbf{m}(t)$. In practice, we alternatively resort to a delay *lookup* table to empirically measure the delay $g(\lambda(t),\mathbf{m}(t))$. Such lookup tables have been used in reinforcement based learning and proven to be effective [13]. We create a table whose row and column indexes indicate workload arrival rates and scaling decisions, respectively, and whose entries are the corresponding delay performance. The entries can be populated with some initial estimates (e.g., based on queueing-theoretic models [14]) at the beginning and then updated in runtime (e.g., using weighted linear regressions) to reflect more accurate delay performance. We discuss how to build such lookup table by offline calibration (Section V-A) and online learning (Section VI-B). Moreover, BATS takes delay lookup table as inputs, and it can be incorporated with any other techniques of estimating delay performance.

B. BATS

Now, we present our online autoscaling algorithm, BATS. Note first that in order to optimally autoscale VM instances (i.e., **P1** formulated in Section II-B), complete offline information is required such that the long-term but limited budget can be optimally split across the entire budgeting period, since otherwise performance may be significantly degraded. For example, if more VM instances are requested in the current time slot, less budget is available for future workloads which may increase dramatically. Accurate prediction of such long-term future workload information is quite challenging and sometimes even impossible, considering possible traffic spikes [3], [15]. Hence, autoscaling decisions need to be made online without *a priori* knowledge of long-term future workloads. To circumvent this practical challenge, we leverage the sample-path Lyapunov technique [16] to develop BATS.

The key idea is that we make autoscaling decisions using a feedback mechanism to meet the desired long-term budget constraint. Specifically, we use the budget deficit at runtime as the feedback information: if there is a temporary budget deficit (e.g., due to high workload), we consider both reducing the expenses and managing the delay, so that the budget deficit can be reduced/eliminated eventually. Otherwise, we focus purely on performance optimization. Mathematically, to track the runtime budget deficit, we construct a (virtual) budget deficit queue which, starting with $q(0)=0$, evolves as follows

$$q(t+1) = \{q(t)+c(t)-z(t)\}^+, \tag{8}$$

where $c(t)$ is the cost at time t, $q(t)$ is the budget deficit queue length and $z(t)$ is the reference budget for time slot t. The reference budget $z(t)$ is not *enforced* as a constraint for the allowed budget over time t; instead, it merely indicates how much money we plan to spend for time t. For example, we can evenly divide the total budget by the total number of budgeting days and obtain daily reference budget, which can be further split to each hour based on the workloads during the prior day. The selection of reference budget $z(t)$ has a negligible impact on the delay performance under common choices, which we verify by our empirical results in Section VI-D.

The budget deficit queue length indicates the deviation of the current cost from the reference budget. Intuitively, with a larger budget deficit at runtime, the autoscaler needs to request fewer VM instances to mitigate the budget deficit at later times for meeting the long-term budget constraint. Thus, the queue length can be leveraged to indicate how much weight we want to give to cost minimization compared to performance optimization, when making autoscaling decisions. To reflect this intuition in our autoscaling decisions, instead of optimizing the delay performance objective in **P1**, we choose to minimize $V\cdot\lambda(t)\cdot g(\lambda(t),\mathbf{m}(t)) + q(t)\sum_{j=1}^{J} p_j(t)\cdot m_j(t)$, where we make decisions in an *online* manner based only on the current workload arrival rate, the budget deficit queue length, and a delay-cost parameter V (which we discuss shortly after). BATS follows the principle **"if exceeding budget, then reduce cost,"** by tracking the budget deficit at runtime and using it as the feedback information to indicate the relative weight/importance of cost minimization versus performance optimization when making autoscaling decisions. The complete description of BATS is provided in Algorithm 1.

C. Analysis

The following theorem proves the performance of BATS.

Algorithm 1 BATS

1: Input $\lambda(t)$ (and $\mathbf{p}(t)$ if applicable), at the beginning of each time slot $t = 0, 1, \cdots, K-1$

2: Choose $\mathbf{m}(t)$ subject to Eqn. (4), (5), and (3) to minimize

$$\textbf{P2}: V \cdot \lambda(t) \cdot g(\lambda(t), \mathbf{m}(t)) + q(t) \sum_{j=1}^{J} p_j(t) \cdot m_j(t) \quad (9)$$

3: Update $q(t)$ according to Equation (8).

Theorem 1. *Suppose that the instantaneous workload arrival rate and delay performance are perfectly known. Then, for any $T \in \mathbb{Z}^+$ and $H \in \mathbb{Z}^+$ such that $K = HT$, the following statements hold.*

a. *The budget constraint is* approximately *satisfied with a bounded deviation:*

$$\frac{1}{K}\sum_{t=0}^{K-1} c(t) \leq \frac{B}{K} + \frac{\sqrt{C(T) + \frac{V}{H}\sum_{h=0}^{H-1}\left(G_h^* - d_{\min}\right)}}{\sqrt{K}}, \quad (10)$$

where $C(T) = U + D(T-1)$ with U and D being finite constants. G_h^* is the minimum delay achieved over $t = (h-1)T, \cdots, hT-1$ by the optimal offline algorithm with T-slot lookahead information over $t = (h-1)T, \cdots, hT-1$, for $h = 0, 1, \cdots, H-1$, and $d_{\min}$ is the minimum delay given in Equation (3)*.*

b. *The delay $\bar{g}^*$ achieved by BATS satisfies:*

$$\bar{g}^* \leq \frac{1}{H}\sum_{h=0}^{H-1} G_h^* + \frac{C(T)}{V}. \quad (11)$$

Proof: *Please see our tech-report [17] for the full proof.*

Theorem 1 provides the *worst*-case performance bound compared to a family of offline algorithms parameterized by their lookahead capabilities characterized by T (i.e., a larger T means the lookahead algorithm looks further into the future). The theorem shows that **BATS achieves delay close to offline optimal while approximately satisfying the budget constraint given arbitrary workload arrivals.** The approximate satisfaction of budget constraint in (10) stems from the fact that workloads may be persistently high: budget may be violated in order to satisfy high workloads, even though budget constraint is satisfied in prior time slots.

Delay-cost parameter V. As formalized in Theorem 1, the delay-cost parameter V of BATS presents the tradeoff between delay performance optimization and budget satisfaction. When V becomes larger, BATS tends to minimize the delay, while giving less attention to the incurred cost, because delay carries more weight on the optimization objective (at Eqn. (9)). Thus, an appropriate selection of V is crucial, but such a value is hard to decide without knowing complete offline information [16]. To address this practical issue, we propose to dynamically update V as follows:

$$V_{new} = \max\{V_{old} + \beta \times \left[z(\bar{t}) - c(\bar{t})\right], V_{\min}\}, \quad (12)$$

where $V_{\min}$ is a sufficiently small positive number to ensure positive V_{new}, β is a positive factor indicating learning rate, and $c(\bar{t}) = \frac{1}{t}\sum_{j=1}^{t} c_j(t)$ and $z(\bar{t}) = \frac{1}{t}\sum_{j=1}^{t} z_j(t)$ are the cumulative average cost and reference budget per slot up to time t, respectively. The intuition for using Eqn. 12 to update V is as follows: if there is a budget deficit over quite a few time slots (i.e., cumulative average cost exceeds average reference budget up to time t), then we have high confidence that V needs to be reduced to place more emphasis on cost minimization such that the long-term budget constraint can be satisfied; and vice versa. In Section VI-C, we show that the desired V can be dynamically found using Equation 12.

Complexity. While **P2** in Algorithm 1 involves integer programming (i.e., autoscaling decision $\mathbf{m}(t)$ can only take integer values), we note that BATS is practically realizable because there are only a reasonably small number of VM types and autoscaling decisions are made only once every time slot. Specifically, given a limit of M on the number of purchased VM instances and J types of VM instances, the worst-case complexity is M^J, which is practically affordable (e.g., $M = 20$ and there are four basic types of VM instances in Azure). Moreover, for many applications, there is usually only one type of VM instances that are the most cost-effective (i.e., provides the best performance given the same cost) [18], reducing the complexity from exponential to linear. In our experiments in Section V, the computation time BATS spent on calculating the decision is just in the order of milliseconds, while scaling decisions are often made in the order of minutes or hours.

IV. System Implementation

This section presents the system implementation of fully-automated BATS autoscaler service on Windows Azure. Our service autoscales VMs running Azure applications, conveniently, effectively and reliably. It provides a graphical user interface (GUI) for users to provide the cost and performance requirements of their applications. Users specify the budgeting period and the total budget as their cost requirement; they also specify their performance requirement: the maximum tolerable application delay d_{max} and the desired delay d_{min}, as illustrated at Equation 3.[1] Once BATS is configured, it autonomously monitors user application and dynamically scales VM resources.

A. Scaler and Monitor

Fig. 1 presents the software architecture of BATS, consisting of three main modules – *Monitor*, *Scheduler* and *Scaler*. *Monitor* gathers different performance metrics of the hosted cloud application and provides the data to *Scheduler*. In Azure, a cloud application writes its performance metric values to a specified Azure table storage periodically, which *Monitor* accesses to collect performance information of the application. To model performance of different applications, *Monitor* supports using different performance metrics, such as CPU usage, memory usage, and network traffic. For example, for a CPU-intensive application, monitoring CPU usage could be more appropriate than monitoring network traffic.

[1] Note that if inappropriately set (e.g., budget is too small), the budget and d_{max} may not be achievable at the same time. One approach to avoid such inappropriate settings is that BATS provides guidance to cloud users based on history data. For example, the minimum required budget can be calculated at the beginning of a budgeting period based on the previous workloads and the tolerable delay performance given by the user. A warning will be prompted if the user provides a budget below the required budget.

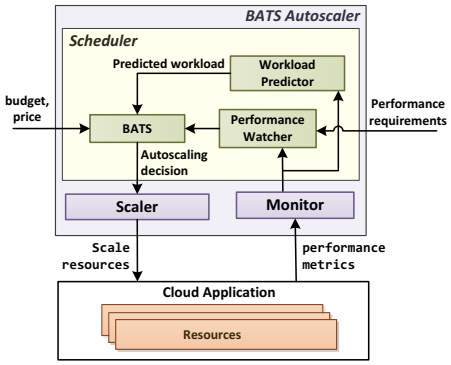


Fig. 1. Architecture diagram of BATS autoscaler.

Scaler executes the scheduler's decision and submits the scaling request to the cloud. It handles the underlying details of connecting to the cloud, certificate management and service status information retrieval. For example, Azure provides a service management API to control the cloud resource configuration. In order to use this API, the users need to create a management certificate in the cloud portal and import it through the GUI of BATS. Whenever *Scheduler* issues a scaling decision, *Scaler* creates and uploads the new VM configuration using the management certificate. Then, Azure scales the VM instances for the hosted application accordingly.

B. Scheduler

Scheduler is the core of our autoscaler service, which consists of three sub modules — *performance watcher*, *workload predictor* and *BATS algorithm*. *Scheduler* operates in both proactive and reactive manner. Its *proactive* behavior is implemented at BATS algorithm, which determines the VM scaling decision at the beginning of each time slot by considering the estimated workload given by workload predictor. The *reactive* behavior takes runtime performance feedback into consideration that is implemented by performance watcher, handling workload prediction inaccuracy and burstiness.

Workload predictor predicts the upcoming workload by analyzing the past values. As a key advantage, BATS does not require long-term workload prediction, which has poor accuracy in practice. Instead, BATS only needs workload prediction for the next time slot. Since the prediction model is not a contribution of this paper and many other prediction models exist [19], [20], [21], we choose to implement the auto-regressive model of [12]. This model predicts $f(d,t)$, the value of a chosen metric at time t of day d, by taking the moving average of the previous N days at the same time t. Mathematically, the predicted value of a metric f at time t is given by $f(d,t) = \frac{1}{N}\sum_{i=1}^{N} a_i * f(d-i,t) + c$. The parameters a_i and c are calibrated online using history data. We discuss the sensitivity of BATS to prediction inaccuracy in Section VI-D.

Performance watcher monitors two types of events continuously: (1) the current workload arrival and (2) the delay status. If there is a significant difference between the predicted workload and observed workload or if the maximum delay cap d_{max} is violated, performance watcher triggers BATS algorithm module to recalculate the scaling decision for current time slot.

BATS implements the algorithm presented in Section III, which determines the VM scaling decision based on the predicted workload arrival before each time slot begins. As it takes about 10 minutes to acquire new VMs in Azure, BATS submits the proactive scaling decision 10 minutes before each time slot begins. In addition, it takes runtime feedback from performance watcher and recalculates the desired VM scaling decision in the event of mis-predicted and/or bursty workloads.

V. Experiment

This section presents experimental results of using BATS to autoscale VM instances hosting a RUBiS web application on Windows Azure. Our results show that BATS achieves up to 34% less average delay compared to the algorithm that evenly divides users budget over all time slots. BATS also reduces users cost by 10% while achieving less delay compared to widely-used reactive scaling rules. Moreover, the performance of BATS is very close to that of the optimal offline algorithm that knows complete future information.

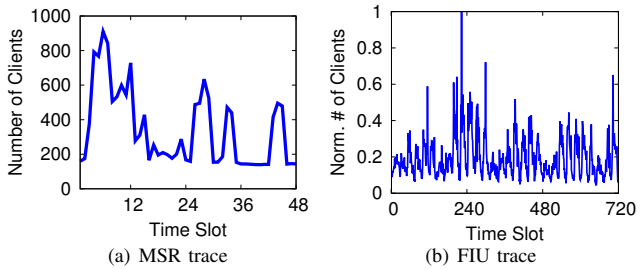
A. Experimental Setup

We deploy RUBiS web application, which implements the core functionality of an auction site: selling, browsing, and bidding. RUBiS is widely used to evaluate the performance and scalability of application servers and virtualized environments [9]. It follows a three-tier web architecture: a front-end web server tier, business logic tier and back-end database tier. We run RUBiS on Windows Azure Cloud Services, which provides a Platform-as-a-Service (PaaS) environment for hosting multi-tier scalable web applications [10].

System configurations. We deploy RUBiS PHP web and business logic tier as scalable PHP web-role on Azure Cloud Services. We deploy the back-end database tier on Azure SQL server which is not scalable. Our experiments scale the VM instances of PHP web-role in the range of 1 to 20, where 20 is the default limit set by Azure [5]. We choose to use extra-small VMs only because they offer the most cost-effective way to execute RUBiS workload. For example, the price of a small VM is 4 times of the extra-small VM, but its throughput is only 3 times, while the larger VMs are even worse in terms of the cost effectiveness. To avoid excessively long experimentation time, the budgeting period in our study consists of 48 time slots and the duration of each time slot is 1 hour. In our experiment, the budget is \$8.5, cost for one VM instance per hour is \$0.02, and the value of delay-cost parameter V is 0.4. We set the desired average delay $d_{min} = 520$ ms, and the maximum tolerable average delay $d_{max} = 1500$ ms.

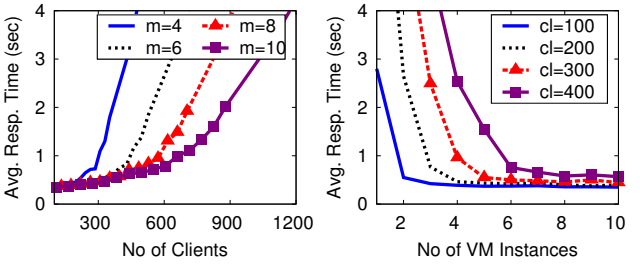
Workload. We use RUBiS workload generator to send client requests to the cloud-end servers. The workload generator creates user sessions (a.k.a clients) which simulates the browsing of an auction site like eBay. The number of clients indicates the amount of workload being generated for the target web site. The average execution time for each web page varies based on the underlying computation. We generate workload arrivals (Fig. 2(a)) based on a workload trace extensively used in prior work [3], representing the activity trace of a few thousand users on enterprise file servers at Microsoft Research.

Autoscaler inputs. To enable autoscaling for RUBiS, our autoscaler uses the number of web connections to monitor



(a) MSR trace (b) FIU trace

Fig. 2. Workload traces.



(a) (b)

Fig. 3. RUBiS load response time correlation. m and cl denote the number of VM instances and clients respectively.

incoming workload. We model the load delay correlation of RUBiS workload using a delay lookup table, which is built at calibration phase by varying the number of clients from the workload generator and obtaining delay for each different scaling configuration (i.e., number of VM instances). Lookup table has been used by prior studies in reinforcement based learning and yields satisfactory performance [13]. We discuss how to build the delay lookup table online in Section VI-B. Fig. 3(a) shows the average delay under different numbers of clients: average delay increases slowly up to certain load, and then it increases exponentially at heavy load (i.e., saturated). Fig. 3(b) shows that if we increase the number of VM instances, average delay decreases down to around 400 ms, after which it becomes almost constant as the average delay is dominated by the request execution time and there is no further delay reduction even if we add more VMs.

B. Experimental Results

We conduct three sets of experiments: (1) compare BATS with three well-known autoscaling algorithms and offline optimal; (2) show the impact of user budget on the performance of BATS; and (3) show the delay-cost tradeoff.

1) Comparison with other autoscaling algorithms

As BATS is the first autoscaling system that incorporates long term budget constraints for interactive web services, there are no directly comparable algorithms. However, to show the effectiveness of BATS, we compare its performance with the following three online autoscaling algorithms and the optimal offline algorithm.

• **EqualSC:** The algorithm evenly divides the available budget across all the time slots and obtains the number of VM instances that can be reserved for the entire budgeting period based on discounted pricing (for reserved instances). We use 20% discount as offered by Windows Azure [5].

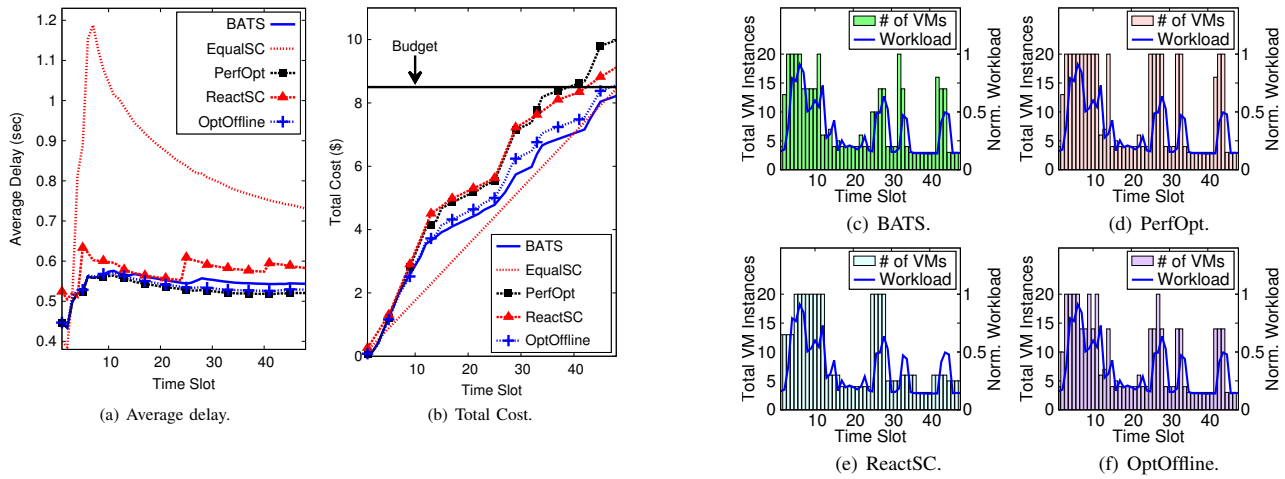
• **ReactSC:** Reactive scaling rules are widely adopted by both developers and third-party solution providers [5], [4], [23], [24], [25]. Most reactive scaling rules are defined by comparing a performance metric to a specific threshold. For example, add a new VM instance when the average CPU utilization exceeds 85% , or terminate a VM instance when the average CPU utilization falls below 45%. We implement a reactive autoscaler that can use more complex rules rather than simple threshold-based comparisons. The autoscaler constantly monitors the average workload arrival rates measured over the last 5 minutes. Then, based on the monitored workload, ReactSC uses the delay lookup table to determine the minimum number of VMs so the resulting delay is equal to the desired delay d_{min} for the upcoming time slot.

• **PerfOpt:** This algorithm knows (using perfect short-term prediction) the workload arrival rate at the beginning of each time slot and uses the delay lookup table to determine the minimum number of VM instances such that the resulting delay is equal to the desired delay d_{min}. It always optimizes performance while disregarding the desired budget constraint. Compared with ReactSC and BATS, PerfOpt assumes perfect short-term prediction information.

• **OptOffline:** The optimal offline algorithm has the perfect workload arrival information for the entire budgeting period at the very beginning. Based on complete offline information, the whole budget is optimally divided among time slots by solving **P1** based on Lagrangian technique and choosing (through bisection search) the optimal Lagrangian multiplier to ensure equality for budget constraint [26]. Essentially, the optimal Lagrangian multiplier corresponds to the budget deficit queue, but it is a fixed value, which can only be obtained based on complete offline workload information. OptOffline is not possible to implement in practice. It only serves as a reference of theoretical optimal.

Firstly, we compare BATS to EqualSC. Figs. 4(a) and 4(b) compare the cumulative average delay and cumulative cost of BATS, respectively. The cumulative average for a time slot t is the corresponding average value of time slot 0 to t. As shown in Fig. 4(a), BATS reduces delay by 34% compared to EqualSC while achieving the same budget constraint, even though EqualSC receives discounted pricing. This is mainly because EqualSC evenly divides the budget across each time slot and reserves 11 VM instances without considering the workload variation. As a result, when there is a workload spike (e.g., in the 4^{th} time slot), the delay becomes very large.

Secondly, we compare BATS with the widely-adopted ReactSC autoscaling mechanism. Fig. 4(a) shows that the average delay reduction of BATS is 10% compared to ReactSC. The degrading performance of ReactSC comes from the long *lagging* time: it takes up to 5 minutes to detect the system status change (e.g., workload variation) and even after detection, it takes up to 10 minutes to acquire a new VM instance. During the lagging time, all the incoming workloads experience longer average delays. For example, ReactSC experiences higher delay in the 25^{th} and 42^{nd} time slots because of its inability to cope instantaneous traffic spikes, as shown in Fig. 4(e). This demonstrates the importance of proactively predicting the



(a) Average delay. (b) Total Cost. (c) BATS. (d) PerfOpt. (e) ReactSC. (f) OptOffline.

Fig. 4. Comparing BATS with other algorithms.

near-future (e.g., hour-ahead) workloads as used by BATS, thereby highlighting the limitations of reactive autoscaling rules. When using ReactSC, keeping an additional resource margin/headroom (i.e., requesting more VM instances than needed) may mitigate temporal excessive delays, but will result in an even higher cost and more likely violate the budget constraint. Moreover, Fig. 4(b) shows that the cost saving of BATS is 10% compared to ReactSC. It is mainly because ReactSC ignores the budget constraint and always makes scaling decisions such that resulting average delay equals d_{min}. This shows that BATS outperforms ReactSC in terms of both delay reduction and cost savings.

Finally, we compare BATS with PerfOpt and OptOffline. Fig. 4(a) shows that although PerfOpt takes 4.4% lower average delay, its resulting cost is 16.8% higher than the user specified budget. This is mainly because PerfOpt only focuses on minimizing the delay without considering budget constraint. The achieved delays of BATS and PerfOpt are 543ms and 520ms, respectively. The additional 23 ms delay does not change the human perception of the web performance, as shown in prior study [27]. In summary, BATS satisfies the budget constraint while achieving a similar delay performance compared to PerfOpt. Furthermore, Fig. 4(a) shows that the average delay of BATS is very close to OptOffline (with a difference less than 4%), while Fig. 4(b) shows that the cost is almost the same. The results demonstrate the effectiveness of BATS: it performs almost as well as the optimal offline autoscaler that requires the complete future prediction.

2) Impact of user budget

We study how user budget affects the behavior of BATS and other benchmarks described earlier. Results show that the delay produced by BATS is never more than 10% compared to that of OptOffline for different users budget, and it is always smaller than EqualSC. We do not show the results of ReactSC and PerfOpt since they are budget-unaware and their performance is independent of the user budget. We first describe the choice of our budget amount that will be used to benchmark the comparison between BATS and other algorithms. The highest

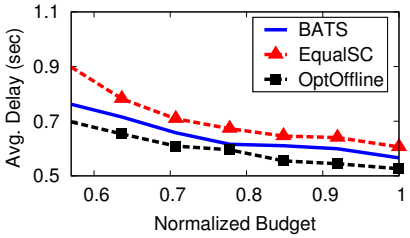


Fig. 5. Impact of users budget.

budget value is chosen based on the incurred cost of PerfOpt (i.e., always selecting the minimum number of VMs such that resulting delay for each time slot is no greater than the predetermined delay threshold d_{min}). We normalize the actual budget by dividing it by the highest budget.

We discuss the impact of user budget with three observations from Fig. 5. (1) The less budget, the higher delay, which matches our expectation. (2) The delay is reduced more rapidly as the budget increases from 55% to 75%, and the delay reduction slows down with further increase of user budget. Under a low budget, few VM instances are used in most of the time slots, resulting in long request waiting time. The long waiting time can be effectively reduced by adding more VM instances with additional budget. However, when the budget increases further, the waiting time becomes smaller and the request execution time dominates the total delay — the delay reduction by adding more VMs becomes smaller. (3) The delay of BATS is not more than 10% compared to the offline optimal, and it is always less than that of EqualSC, showing robust performance of BATS for all budget levels.

3) Delay-cost tradeoff

This experiment discusses how the value of delay-cost parameter V affects BATS in terms of the delay-cost tradeoff under various normalized budget constraints (indicated in the parenthesis to right of "BATS" in Fig. 6). The budget is normalized with respect to the cost of PerfOpt. The result in

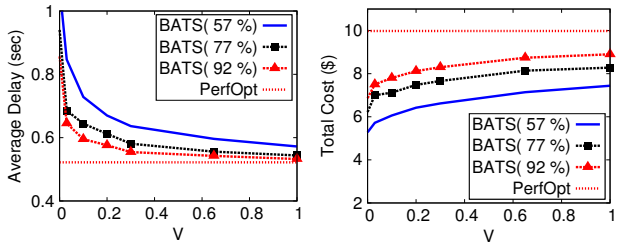


Fig. 6. Impact of V on the delay and total cost.

Fig. 6 is consistent with our analysis: with a greater V, BATS tends to minimize the delay and becomes less concerned about the total cost. The weight of the budget deficit queue becomes less effective. If V goes to infinity, BATS becomes purely performance-driven to minimize the average delay while ignoring budget constraint, and hence reduces to budget-unaware PerfOpt. As a result, the average delay becomes d_{min}. With a smaller V, the budget deficit queue plays a more important role and BATS cares more about the cost. Fig. 6 shows that for very small V, the delay becomes very large (close to d_{max}) and the cost is even less than the specified budget. Note that the delay of PerfOpt represents the minimum delay d_{min} specified by the user. As shown in Fig. 6, when $V \geq 0.4$, the average delay achieved by BATS is fairly close to d_{min}, while still satisfying the budget constraint. At this point, BATS perfectly balances between performance and budget constraint. Section VI-C shows how BATS adapts V autonomously.

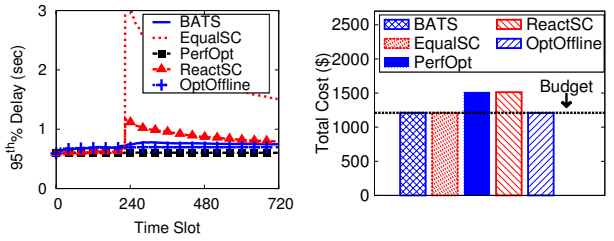
VI. Simulation Results

This section presents simulation results of BATS, which complement the implementation results and evaluate other important aspects of an autoscaling algorithm. We first show the effectiveness and efficiency of BATS by scaling a workload requiring a few hundred VMs. Next, we demonstrate that BATS builds and adapts the delay lookup table and chooses V autonomously in an online manner, without requiring user inputs and with negligible impact on performance and cost.

Simulation setup: We develop a discrete-event based simulator using CloudSim [28] that supports modeling and simulation of virtualized environments. We create a virtualized data center, where each server has 6 CPU cores and 16GB of RAM. Each VM has one core and 1024MB of RAM. Our simulator has a workload generator that mimics the RUBiS workload generator. We evaluate BATS using the workload trace from Florida International University, shown in Fig. 2(b). We obtain this trace by profiling the web server usage logs from January 1 to January 31, 2012. We set the desired average delay $d_{min} = 400$ ms, and the maximum tolerable average delay $d_{max} = 1500$ ms. The cost per VM is \$0.02. We simulate a budget period of 1 month, and a total budget of \$764.

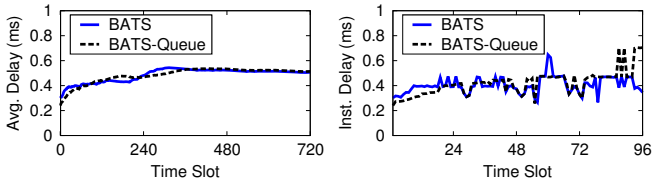
A. Optimizing Average and Tail Delay

We compare BATS to the benchmark algorithms defined in Section V-B1. The results are rather consistent with the implementation results in Fig. 4, and thus we skip the detailed discussion. We omit the average delay results and compare the performance of BATS in terms of 95^{th}-percentile delay.



(a) Cumulative delay. (b) Budget violation.

Fig. 7. Tail delay comparison with other algorithms.



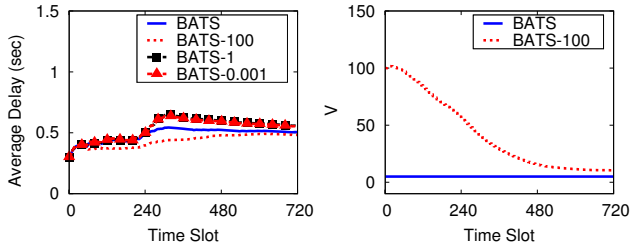
(a) Average delay. (b) Instantaneous delay.

Fig. 8. Delay performance with adaptive delay lookup table.

Here we set $d_{min} = 600$ ms and $d_{max} = 1500$ ms, both in terms of 95^{th}-percentile delay. Fig 7 shows that BATS consistently outperforms the 3 online algorithms and achieves close-to-optimal performance while satisfying budget constraint. The 95^{th} percentile delay of BATS is 96% and 5% lower than EqualSC and ReactSC, respectively. ReactSC and PerfOpt violate the budget constraint and incur 24% more cost than the specified budget. These results show that **beyond average delay, BATS also effectively reduces tail delay**. Furthermore, we use BATS to solve problems with a **few thousand VMs**, and it takes < 50ms to compute the allocation at each time slot, demonstrating its scalability on solving large problems.

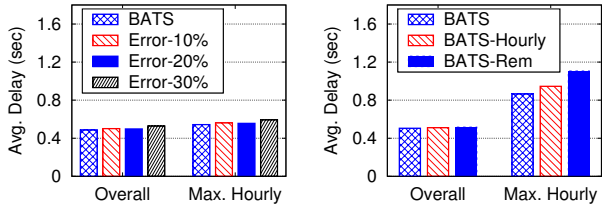
B. Learning Delay Lookup Table Online

To populate initial values in the delay lookup table, we use queueing-theoretic models as a good *approximation* for characterizing delay [3], [14]. For example, we can approximate the VM service process as an M/M/1 queue, for which average delay only depends on two inputs: (1) service rate, i.e., the number of requests that can be processed by a VM in a unit time; and (2) request arrival rate. In our study, we obtain the service rate by pre-running the cloud service on a VM for a short period of time and measuring the saturated throughput under heavy loads. Then, the delay lookup table is fulfilled where each element corresponds to a different combination of arrival rate and number of VM instances. At runtime, the table is updated continuously using the observed delay, and thus it captures the dynamic delay behavior of the web application may change based on the workload mix (e.g., read, or write intensive). Fig. 8(a) shows that the average delay of starting from a delay lookup table initially populated with an M/M/1 queueing model (BATS-Queue) is only 1.5% higher than that of BATS, while satisfying the budget constraint. By using this simple approach to update lookup table online, BATS learns the delay values autonomously and adapts to workload mix during the budget period. There are alternative approaches to building delay lookup tables, e.g., [29] exploits various



(a) Average delay. (b) BATS-100.

Fig. 9. Adaptive V.



(a) Workload prediction error. (b) Reference budget $z(t)$

Fig. 10. Sensitivity study.

techniques offline and online to estimate performance models that relate load and resources with application performance, which are complementary to our work.

C. Choosing Delay-Cost Parameter V Autonomously

To evaluate the effectiveness of autonomously updating the delay-cost parameter V as proposed in Eqn. 12 for , we introduce a variant of BATS algorithm called BATS-x which starts with an initial V value of x. The adaptation rate β in Eqn. 12 is set to 15, and V is adjusted after every 6 time slot. Fig. 9(a) shows that even if BATS starts with a very large or small value of V, it gradually converges and satisfies budget constraint. For example, the desired value of V for this workload is around 5, which can be measured empirically. Fig. 9(b) shows that when BATS-100 starts with an initial V of 100, it self-adapts and eventually becomes close the desired V value. As shown in Fig. 9(a), the corresponding delay till 360^{th} time slot is less than that of BATS because of higher V values. During these time slots, the cost of BATS-100 is higher than the reference budget. However, the V of BATS-100 decreases until average cost per slot becomes higher than average allocated budget per slot. Thus BATS-100 can dynamically adjust V without requiring any user input. While adapting V, the delay of BATS-100 for the whole budgeting period is only 3.6% higher than that of BATS. We also study the behavior of adaptive BATS in case a user starts with a very small value of V. Fig 9(a) shows the average delay of BATS-0.001 for the whole budgeting period is 12% higher than BATS. These results show that because BATS dynamically adapts V, it is robust to its initial parameter settings.

D. Sensitivity Study

Prediction errors: As BATS leverages the hour-ahead workload prediction, we evaluate how BATS performs in the presence of prediction errors. We consider four cases where the workload predictor introduces prediction errors of 0%,$\pm$10%, $\pm$20% and $\pm$30%. Fig.10(a) shows that, compared to 0% prediction error, the resulting delay increases by 0.9%, 2.8% and 12.3% for $\pm$10%, $\pm$20% and $\pm$30% prediction errors, respectively. Even with high prediction errors, the performance of BATS is still quite robust. Moreover, in practice, the reactive part of BATS also compensates for the prediction error and ensure that the delay does not violate user requirements. Thus, BATS can be successfully applied even when the workload prediction is not perfect.

Reference budget $z(t)$: We explore the impact of different choices of reference budget $z(t)$ in Eqn. 8. In our above studies, the reference budget $z(t)$ at time slot t in BATS is obtained by dividing the whole budget evenly to each time slot. For comparison purposes, we consider two variants of BATS, where we set reference budget $z(t)$ by: (1)**BATS-Rem:** evenly dividing the remaining budget at time slot t; (2) **BATS-Hourly:** dividing the budget to each time slot according to the average workload arrival rate obtained from past data. Fig. 10(b) shows that while choosing $z(t)$ differently, the delay performance remains relatively the same with less than 2% difference. Intuitively, the reference budget $z(t)$ in Eqn. 8 only directly impacts the runtime budget deficit queue dynamics, thereby not being enforced as runtime budget constraint or directly impacting the autoscaling decision. In the long term, as long as the total budget is the same, $z(t)$ has a negligible impact on the delay performance, demonstrating the robustness of BATS against different choices of $z(t)$.

VII. Related Work

Autoscaling. In recent years, autoscaling has become an integral feature of cloud computing, and various autoscaling mechanisms have been proposed to enable elastic resource acquisitions for performance and cost effectiveness. In general, autoscaling techniques can be classified as "proactive" and "reactive". In "proactive" autoscaling, decisions are actively triggered by users via, e.g., predictive modeling [30], [31], [32], [33], whereas in "reactive" autoscaling, decisions are made in passive response to system statuses (e.g., CPU utilization) [34]. For example, [30], [31], [32] use prediction/learning techniques to estimate workload demand/arrival rates for autoscaling, while [33] builds a performance model to make autoscaling decisions. Many cloud service providers offer both schedule-based and rule-based "reactive" autoscaling [5], [4], [23], [24], [25]: cloud users can specify customized schedules to initiate/release VM instances at particular times using schedule-based autoscaling, while rule-based autoscaling scales VM instances based on resource usage thresholds (e.g., CPU, memory usage). BATS exploits benefits of both proactive and reactive scaling. Primarily, we scale resources proactively based on deficit queue length and short-term workload prediction, while we also incorporate reactive autoscaling as a backup during exceptions (e.g., workload spikes).

There have been some prior studies on satisfying *short-term* budget constraint. For example, [6] uses a constant hourly budget to decide optimal number of VM instances for jobs that have larger deadline (e.g., 1 hour), while [18] also scales and schedules cloud workflows considering the hourly budget constraint for each individual job. Similarly, [7],

[8] optimizes workflow scheduling by exploiting flow-specific properties (e.g., user-specified priorities) while considering an instantaneous budget constraint. These studies only impose a short-term (e.g., hourly) budget constraint, which bounds resource availability at each step independently. Our work considers an even harder problem: the resource availability over different time steps is dependent as we bound the total resources across the entire budgeting period.

Resource management and long-term constraint: Our work broadly lies in the category of dynamic resource management and hence is also related to a number of other domains, such as server management in data centers [15], [35]. While many efforts have been dedicated to enable autonomic and self-managing systems using control theoretic and learning approaches [35], [16], [36], only a few address long-term performance/constraints. For example, the existing research that deals with long-term constraints (e.g., brown energy [37], monthly cost [38]) in data centers often relies on accurate predictions of future information that may not be available in practice. To the best of our knowledge, we develop the first provably-efficient online autoscaling solution to optimize delay performance for real-world cloud applications while satisfying long-term budget constraint.

VIII. Conclusions

This paper provides a full-fledged autoscaling solution, BATS, to optimize delay performance while meeting users' long-term budget constraints using only past and instantaneous workload information. Analytically, we proved that the autoscaling algorithm of BATS achieves a close-to-optimal performance even compared to the optimal solution that has complete offline information. We implemented BATS autoscaler as an automated service for cloud applications on Windows Azure. We conducted extensive experimental and simulation studies showing the effectiveness, autonomicity, and robustness of BATS on a wide range of scenarios with various workloads.

Acknowledgment

This work was partially done while S. Ren was with Florida International University, and was supported in part by the U.S. NSF CNS-1423137 and CNS-1453491.

References

[1] Uptime Institute, "Data center industry survey," http://uptimeinstitute.com/2013-survey-results, 2013.

[2] S. VanRoekel, "The FY14 President's IT budget: Innovate, deliver, protect," https://cio.gov/the-fy14-presidents-it-budget-innovate-deliver-protect/.

[3] M. Lin, A. Wierman, L. L. H. Andrew, and E. Thereska, "Dynamic right-sizing for power-proportional data centers," in *INFOCOM*, 2011.

[4] "Amazon ec2," http://aws.amazon.com/autoscaling/.

[5] "Windows azure," http://www.windowsazure.com/.

[6] M. Mao, J. Li, and M. Humphrey, "Cloud auto-scaling with deadline and budget constraints," in *GRID*, 2010.

[7] M. Malawski, G. Juve, E. Deelman, and J. Nabrzyski, "Cost-and deadline-constrained provisioning for scientific workflow ensembles in iaas clouds," in *SC*, 2012.

[8] R. Sakellariou, H. Zhao, E. Tsiakkouri, and M. D. Dikaiakos, "Scheduling workflows with budget constraints," in *Integrated Research in GRID Computing*, 2007.

[9] "Rubis: Rice university bidding system," http://rubis.ow2.org/.

[10] "Azure cloud service," http://www.windowsazure.com/en-us/manage/services/cloud-services/what-is-a-cloud-service/.

[11] Z. Liu, M. Lin, A. Wierman, S. H. Low, and L. L. Andrew, "Greening geographical load balancing," in *SIGMETRICS*, 2011.

[12] Z. Liu, Y. Chen, C. Bash, A. Wierman, D. Gmach, Z. Wang, M. Marwah, and C. Hyser, "Renewable and cooling aware workload management for sustainable data centers," in *SIGMETRICS*, 2012.

[13] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani, "On the use of hybrid reinforcement learning for autonomic resource allocation," *Cluster Computing*, vol. 10, no. 3, pp. 287–299, 2007.

[14] N. U. Prabhu, *Foundations of Queueing Theory*. Kluwer Academic Publishers, 1997.

[15] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch, "Autoscale: Dynamic, robust capacity management for multi-tier data centers," *TOCS*, vol. 30, pp. 14:1–14:26, 2012.

[16] M. J. Neely, *Stochastic Network Optimization with Application to Communication and Queueing Systems*. Morgan & Claypool, 2010.

[17] A. H. Mahmud, Y. He, and S. Ren, "BATS: Budget-constrained autoscaling for cloud performance optimization," http://scg.cs.fiu.edu/tech/bats.pdf.

[18] M. Mao and M. Humphrey, "Scaling and scheduling to maximize application performance within budget constraints in cloud workflows," in *IPDPS*, 2013.

[19] Q. Zhang, L. Cherkasova, G. Mathews, W. Greene, and E. Smirni, "R-capriccio: A capacity planning and anomaly detection tool for enterprise services with live workloads," in *Middleware*, 2007.

[20] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper, "Capacity management and demand prediction for next generation data centers," in *ICWS*, 2007.

[21] A. Ganapathi, Y. Chen, A. Fox, R. Katz, and D. Patterson, "Statistics-driven workload modeling for the cloud," in *ICDEW*, 2010.

[22] J. Zhu, Z. Jiang, and Z. Xiao, "Twinkle: A fast resource provisioning mechanism for internet services," in *INFOCOM*, 2011.

[23] "Rackspace," http://www.rackspace.com/.

[24] "Scarl," http://www.scalr.com/.

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

[26] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.

[27] F. F.-H. Nah, "A study on tolerable waiting time: how long are web users willing to wait?" *Behaviour & Information Technology*, vol. 23, no. 3, pp. 153–163, 2004.

[28] R. Buyya, R. Ranjan, and R. N. Calheiros, "Modeling and simulation of scalable cloud computing environments and the cloudsim toolkit: Challenges and opportunities," in *HPCS*, 2009.

[29] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang, "Adaptive, model-driven autoscaling for cloud applications," in *ICAC*, 2014.

[30] Z. Gong, X. Gu, and J. Wilkes, "Press: Predictive elastic resource scaling for cloud systems," in *CNSM*, 2010.

[31] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "Cloudscale: elastic resource scaling for multi-tenant cloud systems," in *SoCC*, 2011.

[32] N. Roy, A. Dubey, and A. Gokhale, "Efficient autoscaling in the cloud using predictive models for workload forecasting," in *CLOUD*, 2011.

[33] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang, "Adaptive, model-driven autoscaling for cloud applications," in *ICAC 14*, 2014.

[34] N. D. Mickulicz, P. Narasimhan, and R. Gandhi, "To auto scale or not to auto scale," in *ICAC*, 2013.

[35] Y. Diao, J. L. Hellerstein, S. Parekh, R. Griffith, G. E. Kaiser, and D. Phung, "A control theory foundation for self-managing computing systems," *JSAC*, vol. 23, pp. 2213–2222, 2005.

[36] P. Shivam, S. Babu, and J. S. Chase, "Learning application models for utility resource planning," in *ICAC*, 2006.

[37] K. Le, R. Bianchini, T. D. Nguyen, O. Bilgir, and M. Martonosi, "Capping the brown energy consumption of internet services at low cost," in *IGCC*, 2010.

[38] Y. Zhang, Y. Wang, and X. Wang, "Electricity bill capping for cloud-scale data centers that impact the power markets," in *ICPP*, 2012.