Coordinating Fast Concurrency Adapting With Autoscaling for SLO-Oriented Web Applications

Jianshu Liu, *Member, IEEE*, Shungeng Zhang, *Member, IEEE*, Qingyang Wang, *Member, IEEE*, and Jinpeng Wei, *Member, IEEE*

Abstract—Cloud providers tend to support dynamic computing resources reallocation (e.g., Autoscaling) to handle the bursty workload for web applications (e.g., e-commerce) in the cloud environment. Nevertheless, we demonstrate that directly scaling a bottleneck server without quickly adjusting its soft resources (e.g., server threads and database connections) can cause significant response time fluctuations of the target web application. Since soft resources determine the request processing concurrency of each server in the system, simply scaling out/in the bottleneck service can unintentionally change the concurrency level of related services, inducing either under- or over-utilization of the critical hardware resource. In this paper, we propose the Scatter-Concurrency-Throughput (SCT) model, which can rapidly identify the near-optimal soft resource allocation of each server in the system using the measurement of each server's real-time throughput and concurrency. Furthermore, we implement a Concurrency-aware autoScaling (ConScale) framework that integrates the SCT model to quickly reallocate the soft resources of the key servers in the system to best utilize the new hardware resource capacity after the system scaling. Based on extensive experimental comparisons with two widely used hardware-only scaling mechanisms for web applications: EC2-AutoScaling (VM-based autoscaler) and Kubernetes HPA (container-based autoscaler), we show that ConScale can successfully mitigate the response time fluctuations over the system scaling phase in both VM-based and container-based environments.

Index Terms—Scalability, auto-scaling, soft resource, cloud-based applications

1 Introduction

For modern cloud platforms, scalability is an important requirement, which enables an application to scale its computing resources under varying workloads dynamically. Such ability is especially meaningful for modern web-facing applications (e.g., e-commerce) due to their naturally bursty workloads [1]. For instance, the number of users visiting the Amazon website over holidays (e.g., Black Friday) can be 10X than that in normal periods [2]. The traditional strategy that always provisions sufficient resources for the peak workload of the system will waste massive amounts of computing resources and power because of low resource utilization (e.g., averagely 18% [3]). Hence, automatically adjusting the scale of a web application system to deal with workload variations is extremely significant.

Effectively scaling a web application is more challenging than parallel batch workloads (e.g., MapReduce or Hadoop) for two reasons. The first reason is that most web applications have strict Quality of Service (QoS) requirements. For instance, web search requires stringent bounded response time (e.g., 99th percentile response time $<$ 300ms [4], [5], [6]). Due to the bursty nature of web application workloads (e.g., Slashdot effect [7]), intelligently scaling the necessary computing resource to adapt to the runtime workload variations and always satisfy the QoS requirement can be difficult. The system will encounter temporary overloading unavoidably even if we apply reactive [8] or proactive [9], [10] autoscaling mechanisms. For instance, Fig. 1 displays the large response time fluctuations of a 3-tier RUBBoS benchmark application (detailed experiment setup is included in Section 5.1) adopting the EC2-AutoScaling mechanism[1] to scale the number of VMs to deal with the bursty workload. We noticed that the temporary overloading during the system scaling phase causes large latency spikes in a real cloud computing environment enabling autoscaling.

Besides adjusting hardware resources, soft resource allocation (e.g., server threads or connections) plays a crucial role in web application performance. Previous research [11] demonstrates that scaling by adjusting the number of running VMs in an n-tier system can unintentionally change the request processing concurrency of individual servers, which can incur either over- or under-utilization of the critical

- *Jianshu Liu and Qingyang Wang are with the Division of Computer Science and Engineering, Louisiana State University, Baton Rouge, LA 70803 USA. E-mail: {jliu96, qwang26}@lsu.edu.*
- *Shungeng Zhang is with the School of Cyber and Computer Sciences, Augusta University, Augusta, GA 30912 USA. E-mail: szhang2@augusta.edu.*
- *Jinpeng Wei is with the Department of Software and Information Systems, University of North Carolina at Charlotte, Charlotte, NC 28223 USA. E-mail: jwei8@uncc.edu.*

Manuscript received 29 June 2021; revised 21 Jan. 2022; accepted 8 Feb. 2022. Date of publication 14 Feb. 2022; date of current version 2 June 2022.
This work was supported in part by National Science Foundation by CISE's under Grant CNS-2000681, in part by the Office of Naval Research under Grant N00014-21-1-2171/N00014-19-1-2371, and in part by Army Research Office under Grant W911NF-17-1-0437, and Grants from Fujitsu.
(Corresponding author: Qingyang Wang.)
Recommended for acceptance by S. Chen.
Digital Object Identifier no. 10.1109/TPDS.2022.3151512

1. EC2-AutoScaling allows users to set a simple resource utilization threshold (e.g., CPU utilization $>$ 80%) for scaling decisions [8].

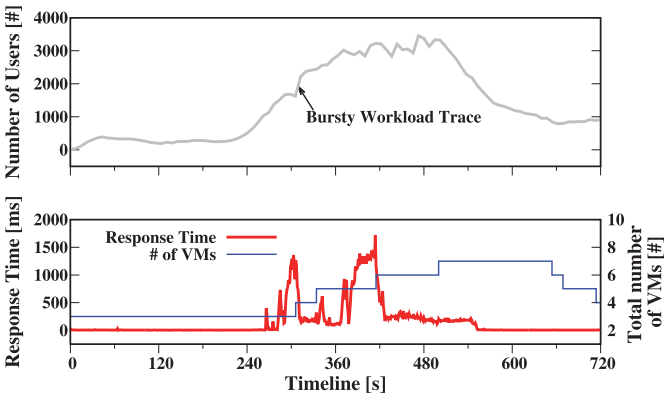


Fig. 1. Large response time spikes of an n-tier RUBBoS application during the system scales (through EC2-AutoScaling strategy) due to bursty workload.

hardware resource of the system [12]. Therefore, without soft resource re-adaption, the hardware-only scaling mechanisms [9], [10], [13] such as EC2-AutoScaling may not make full use of newly added hardware resources. Even for some recent container-based autoscalers designed for monolithic [14], [15] or microservice-based system [16], [17], such uncoordinated scaling mechanisms may still negatively affect the system performance due to the mismatch between the soft and the hardware resources [18]. To handle such a problem, a recent research [11] implements a framework named DCM that applies a concurrency adaption management (through soft resource reallocation) after the hardware resource scaling. Concretely, DCM adopts a trial-and-error offline-profiling approach to determine their concurrency adaption policies before the production phase. However, the static offline profiling approach is very time-consuming since it needs to run extensive experiments to build the performance model. Furthermore, frequent model reconstruction and retraining are required once the production runtime environment conditions (e.g., the critical hardware resource, the system state, and workload characteristic) vary from that in the offline-profiling phase. Thus, offline models cannot provide timely soft resource re-adaption for SLO-oriented web applications that have strict latency requirements (in milliseconds level).

In this paper, we develop an online Scatter-Concurrency-Throughput (SCT) model to rapidly identify the near-optimal concurrency for component servers using the real-time measurement of each server's application-level metrics (e.g., concurrency and throughput). Assuming each server in a web system records the arrival and the departure timestamps of each request at millisecond granularity via a request accessing log, we can measure each server's real-time throughput and concurrency by calculating the request completion rate and the number of concurrent requests within short time intervals (e.g., 50ms), respectively. According to the classic Utilization Law [19], the optimal concurrency setting for a server is the minimum of concurrency when the server reaches the highest throughput. Thus, by correlating the fine-grained throughput and concurrency measured in continuous short time windows, our model can recommend a rational concurrency of each server during runtime.

We implement a Concurrency-aware system Scaling (ConScale) framework, which coordinates the rational soft resource allocation recommended by our SCT model with hardware resource scaling. By analyzing fine-grained measured throughput and concurrency, the SCT model continuously recommends each server's near-optimal concurrency setting on the fly. Specifically, our ConScale framework takes two actions during a system scaling phase: First, adding or removing hardware resources via a classic threshold-based scaling mechanism (e.g., EC2-AutoScaling for VM-scaling or Kubernetes HPA for container-scaling). Second, re-adapting the soft resources of the related server as recommended by the SCT model after hardware scaling. Integrating both critical hardware resource scaling with runtime rational soft resources adapting, our ConScale framework can mitigate the system response time fluctuations during the temporary overloading phases over the system scaling processes.

In brief, our work makes the following contributions:

- Develop the online SCT model, which can rapidly provide the updated optimal concurrency setting of each server in a web system according to runtime environment conditions (Section 3).
- Reveal three factors that can cause the shifting of the optimal concurrency setting of component servers (e.g., Tomcat or MySQL) in a web system (Sections 2.2 and 3.4).
- Implement the ConScale framework to realize fast and intelligent soft resources adaption to handle temporary overloading in system scaling scenarios in clouds (Section 4).
- Conduct an extensive evaluation of the performance of ConScale with a realistic bursty workload for scaling monolithic (i.e., n-tier) and microservice-based applications (Section 5).

We outline the rest of this paper as follows. Section 2 displays that the optimal concurrency for component servers shifts as system condition changes (e.g., the critical hardware resource scaling and system state change). Section 3 presents the online Scatter-Concurrency-Throughput model and our empirical study on factors that may affect optimal concurrency based on the SCT model. Section 4 introduces the design of our ConScale framework and implementation details. Section 5 discusses the experimental evaluation under six categorized realistic workloads. Section 6 summarizes the related work, and Section 7 concludes the paper.

2 Background and Motivation

2.1 Experiment Setup

We use a representative n-tier application benchmark RUBBoS [20], which is one of the bulletin board applications modeled after Slashdot [7]. The RUBBoS benchmark can be configured as 3-tier (i.e., web server tier, application server tier, and database server tier) or 4-tier (add load balancer tier like HAProxy [21] or cache tier like Memcached [22]). The benchmark application consists of 24 servlets such as "StoriesOfTheDay". We use two types of workload generators. First, we use a revised RUBBoS workload generator, which sets zero think time for sending consecutive requests

Software Stack

Web Server	Apache 2.2.31 +tomcat-connecters-1.2.28
Application Server	Tomcat 7.0.65 +mysql-connector-java-5.1.19
Load Balancer	HAProxy 2.0
Database Server	MySQL 5.1.62
Operating System	RHEL 6.10 (kernel 2.6.32)
Hypervisor	VMware ESXi v6.0
JDK Version	Oracle JDK 1.8.0
Docker Engine	Docker 19.03.13
Container Orchestration	Kubernetes 1.14.2

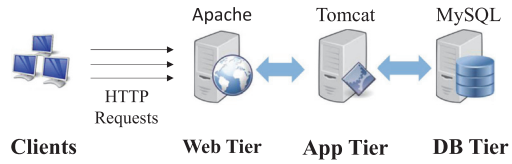
ESXi Host Configuration

Model	Dell Power Edge R430
CPU	2*Intel Xeon E5-2603 v3 1.6 GHz Hexa-Core
Storage	7200rpm SATA local disk
Memory	16GB

VM Configuration

# vCPU	1
CPU limit	1.6GHz
CPU shares	Normal
vRAM	2GB
vDisk	20GB

(a) Software Stack and Hardware Specification.



(b) 1/1/1 Sample Topology

Fig. 2. Experimental setup of the VMware ESXi cluster.

to stress the target server with precisely controlled concurrency in Section 2.2. Second, we use the original RUBBoS workload generator, which simulates a number of concurrent users by generating a request rate that follows a Poisson distribution in Section 5. Both two workload generators provide two types of workload: "browse-only" CPU-intensive and "read/write-mix" I/O intensive workload.

Fig. 2 displays the experimental setup including software stack, hardware specification, and sample topology of our experiments. We conduct our experiments in a private VMware ESXi cluster [23]. We adopt a three-digit notation $\#Web/\#App/\#DB$ to denote the number of servers in each tier (i.e., web tier, application tier, and database tier) in the system. For instance, Fig. 2b displays a 1/1/1 sample topology, referring to one Apache, one Tomcat, and one MySQL. Each individual server is deployed in a virtual machine or a container running in a dedicated physical node in our private cluster. We evaluate three representative soft resources: the thread pool in a web server and an app server, and the DB connection pool[2] in an app server. These three soft resources determine the maximum level of the request processing concurrency in Apache, Tomcat, and MySQL, respectively. We denote such three soft resources as $\#W_{threads} - \#A_{threads} - \#DB_{connections}$. For example, the soft resource allocation can be 100-100-20 in a 1/1/1 hardware topology (see Fig. 2b), which indicates 100 Apache threads, 100 Tomcat threads, and 20 DB connections.

2.2 Shifting of Optimal Concurrency Setting as Environmental Condition Changes

This section shows an experimental study of the shifting of optimal concurrency settings in component servers of a 3-tier system due to environmental condition changes. Industry practitioners [24] and academic researchers [12] commonly adopt a brute-force profiling on various concurrency workloads to identify the optimal concurrency settings. Fig. 3 displays the performance variation at increasing workload concurrency for tuning the optimal Tomcat thread pool allocation. We set the same number of threads in the corresponding server along with the increasing concurrency level to exclude the queue overflow problem [11].

We study two common environmental condition change scenarios in the production phase. First, the vertical scaling for the critical resource (e.g., add/remove # CPU cores for one instance) can affect the optimal concurrency setting. For example, Figs. 3a and 3b show the optimal concurrency setting in Tomcat shifts from 10 to 20 after we manually scale up the Tomcat CPU from 1-core to 2-core. Our experimental results indicate that vertical scaling which is adopted by many autoscalers (e.g., Cloudscale [25] and Kubernetes VPA [26]) would lead to the optimal concurrency setting being sub-optimal after the system scaling.

Second, system state change also incurs the shifting of the optimal concurrency setting. We note that the dataset for web services updates continuously [27], which would cause the service rate variation and further affect the optimal concurrency level of the application server (e.g., Tomcat). Figs. 3b and 3c show that the optimal concurrency setting shifts from 20 to 15 after the dataset is manually enlarged, even though such two cases have the same critical hardware resource (e.g., 2-core CPU) under the same workload. Consequently, the optimal concurrency setting for Tomcat is sensitive to the system state change.

So far our experimental results demonstrate that the environmental condition changes have a considerable impact on the optimal concurrency settings in component servers of a web system. To adapt to environmental condition changes, static optimal soft resource allocation, which is based on offline brute-force search, is very time-consuming due to exhaustive profiling and constant model retraining. Furthermore, considering the unpredictable web system state, dynamic resource arrangement in scaling scenario, and naturally bursty workload of web applications, online optimal concurrency estimation and fast runtime soft resource adapting should be integrated into the web system scaling management design.

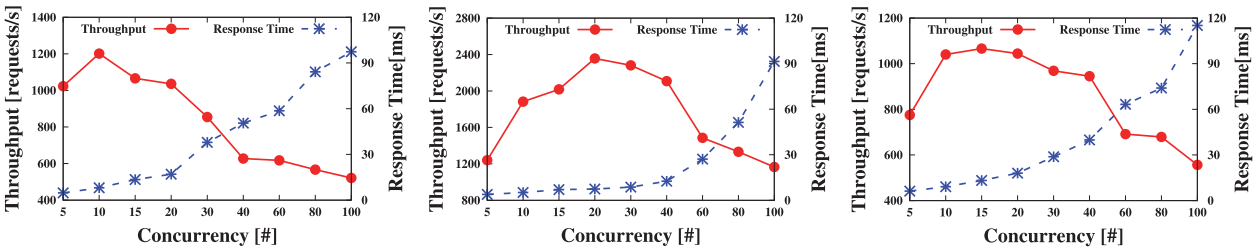
3 Scatter-Concurrency-Throughput Model

We develop an online Scatter-Concurrency-Throughput (SCT) Model which can quickly generate an updated near-optimal concurrency setting for the component server of a web system. Inspired by a statistical intervention analysis for detecting system bottleneck [28], our model extends such analysis by considering the non-trivial multithreading overhead of the server under high request processing concurrency. The objective of our model is to generate the latest near-optimal concurrency setting for each server along with the runtime environmental condition, which can guarantee stable response time together with high throughput at the same time.

3.1 Model Description

According to the classic Utilization Law [19], the server's throughput grows linearly as the workload concurrency increases until the server is saturated. As the workload concurrency continuously increases, the throughput is on a plateau or even decreases because of the non-trivial

2. The maximum request processing concurrency of the DB server is limited by the DB connection pool size in the upstream application server.



(a) Throughput reaches the highest when the request processing concurrency is 10 for 1-core Tomcat.

(b) Throughput reaches the highest when the concurrency is 20 after adding one CPU core (i.e., 2-core Tomcat).

(c) Throughput reaches the highest when the concurrency is 15 for 2-core Tomcat after doubling the dataset size.

Fig. 3. Tomcat performance variation along with increasing request processing concurrency in a 3-tier system. Figs. 3a and 3b show the vertical scaling of critical hardware resource (e.g., # of CPU cores) can change the Tomcat optimal concurrency setting (10 to 20). Figs. 3b and 3c show the system state change (e.g., dataset size increase) can change the optimal concurrency setting (20 to 15) in Tomcat.

multithreading overhead [11], [29], [30], [31]. Meanwhile, the response time increases significantly and causes Service Level Objectives (SLO) violations under high workload concurrency. Therefore, there is a range of rational concurrency settings that contribute to the highest system throughput. Fig. 4 characterizes the correlations among a server's throughput, response time, and concurrency, respectively, which help us further identify the server's rational concurrency range.

We determine the `lower bound` of a rational concurrency setting range (i.e., Q_{lower}) based on correlation between throughput and concurrency. The server throughput makes a linear growth until reaching the highest throughput TP_{max} as workload concurrency increases. The highest throughput depends on the average critical resource expense per request since the server only reaches the highest throughput when its critical resource is fully utilized. Beyond saturation, the server can maintain the maximum throughput (i.e., TP_{max}) along with the concurrency increase. We can tune the Q_{lower} by identifying that the ratio of the increment of throughput approaches to zero. In the meantime, we can achieve the minimum response time under the Q_{lower} concurrency setting.

The server performance would start to degrade finally along with the server concurrency further increases and exceeds the `upper bound` of that server rational concurrency setting (i.e, Q_{upper}). For example, throughput drop along with long response time is widely common in servers (e.g., MySQL) which adopts a thread-based synchronous mechanism. Such type of server arranges a dedicated thread for processing a single request [18]. Previous research studies [11], [29], [30], [31] prove that thread-based servers serving high-level concurrency would present considerable multithreading overhead due to many factors (e.g., thread contention or consistency penalty [11], [31]). Such multithreading overhead caused by too many concurrent requests processing may force a non-linear system throughput drop and long response time. To avoid the SLO violations due to the high concurrency setting, we can specify a latency threshold and tune the corresponding concurrency as Q_{upper} based on the correlation between response time and its concurrency.

Our Scatter-Concurrency-Throughput (SCT) model determines a rational concurrency range (i.e., $[Q_{lower}, Q_{upper}]$) by correlating the runtime metrics measured from an individual server and modeling the relationships between performance metrics (e.g., throughput and response time) and concurrency. Fig. 4 illustrates the workflow of rational concurrency range determination, which consists of two major phases: Real-time Metrics Collection Phase and Rational Concurrency Range Estimation Phase.

Real-Time Metrics Collection Phase. The SCT model collects a series of tuples $\{Q_{tn}, TP_{tn}, RT_{tn}\}$ during a short time period (e.g., 3 minutes). Each tuple consists of a server's real-time concurrency, throughput, and response time measured at a fine granularity (e.g., 50ms). As a server's real-time concurrency under practical workload varies, we denote the server concurrency region during such time period as $[Q_{min}, Q_{max}]$. For certain server concurrency $Q_n (Q_n \in [Q_{min}, Q_{max}])$, we calculate the average throughput $\overline{TP_n}$ and average response time $\overline{RT_n}$ to represent the system performance. After that, we extract the main sequence curve (i.e., blue lines in Fig. 4), which consists of the processed data tuples $\{Q_n, \overline{TP_n}, \overline{RT_n}\}$.

Rational Concurrency Range Estimation Phase. Our SCT model aims to identify the rational concurrency range from the extracted main sequence curve. We apply the statistic intervention analysis [28] to estimate the minimum rational concurrency setting (i.e., Q_{lower}). Moreover, we determine the maximum rational concurrency setting (i.e., Q_{upper}) by referring to the SLO requirements for each application. We are capable of generating the rational concurrency range $[Q_{lower}, Q_{upper}]$ of a typical server as shown in Fig. 4. Considering the strict bounded response time for modern web systems, we select the Q_{lower} as the optimal concurrency setting since we need to guarantee a low response time and make a

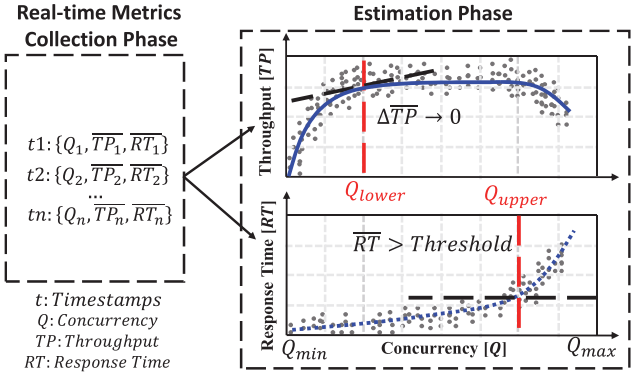
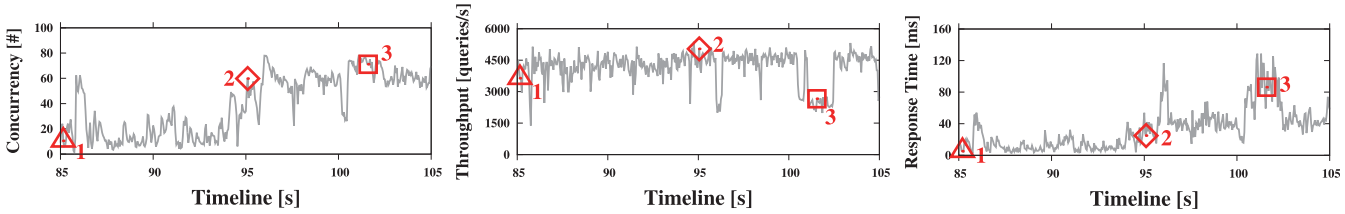


Fig. 4. Illustration of SCT model workflow for rational concurrency range determination.



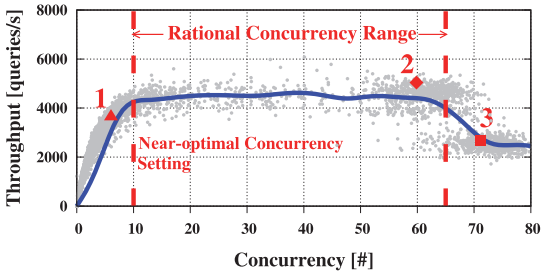
(a) MySQL concurrency. (b) MySQL throughput. (c) MySQL response time.

Fig. 5. Measurement of runtime fine-grained application-level metircs of MySQL in a 3-tier system when facing a realistic bursty workload. Figs. 5a, 5b, and 5c show MySQL's "real-time" (measured at 50ms time granularity) concurrency, throughput, and response time within the same 20-second experiment period, respectively. We mark out three points to match their positions in Fig. 6 for better illustration.

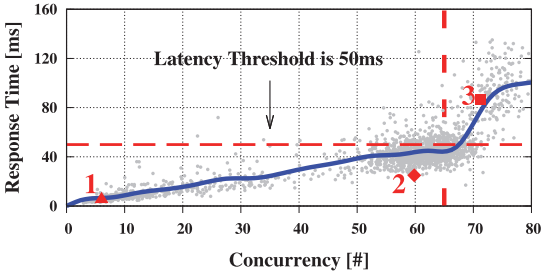
trade-off between throughput and response time. (see the dashed line in Fig. 4).

3.2 Fine-Grained Throughput and Concurrency Correlation

Real-time fine-grained measurement of a server's application-level metrics is one of the key points for correctly characterizing the performance of the component server in a web system under the bursty workload. We calculate the real-time throughput by counting the number of completed requests within a sufficiently short time interval (e.g., 50ms); the real-time response time and concurrency are obtained by calculating the average response time of completed requests and counting the number of concurrent requests within the same time interval, respectively. Figs. 5a, 5b, and 5c show the concurrency, throughput, and response time of a MySQL server after the system scales from 1/1/1 to 1/2/1. The bottleneck in the system shifts from Tomcat to MySQL after adding a new Tomcat. We note that the concurrency, throughput, and response time in MySQL all present large fluctuations after scaling out Tomcat servers, indicating MySQL suffers from less CPU efficiency.



(a) The scatter graph shows the correlation between MySQL throughput and concurrency. We use the blue line as the trend line.



(b) The scatter graph shows the correlation between MySQL response time and concurrency. We use the blue line as the trend line.

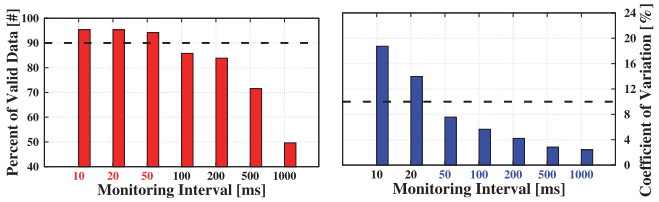
Fig. 6. The correlations between MySQL concurrency, throughput, and response time measured at 50ms granularity during a 12-minute experiment. Through two scatter graphs, the SCT model can generate a rational concurrency range for MySQL. We choose the lower bound of such rational range as the optimal concurrency setting (i.e., 10 in Fig. 6a) because the corresponding throughput reaches the maximum and response time gets to the minimum within the range.

We quantitatively validate that high request processing concurrency causes less CPU efficiency in MySQL through correlating the MySQL's application-level metrics (i.e., throughput, response time, and concurrency) in Fig. 6. Fig. 6a correlates the MySQL concurrency and throughput during a 12-minute experiment, and each point refers to a pair of MySQL concurrency and throughput measured at the same time interval. We also plot Fig. 6b by correlating the pairs of MySQL concurrency and response time. Each marked point (i.e., Point 1, 2, and 3) in Fig. 6 can be localized in Fig. 5. Fig. 6a presents a three-stage pattern for throughput along with concurrency increase, and response time increases significantly after concurrency exceeding Q_{upper} in Fig. 6b, which are consistent with our previous analysis in Section 3.1.

Recall our online SCT model (Section 3.1) on the performance of the component server in the system under a realistic workload, we observe that either too small (e.g., $< Q_{lower}$) or too large (e.g., $> Q_{upper}$) of the request processing concurrency can lead to inferior performance of a server, which is caused by low CPU efficiency and high multithreading overhead. Thus, to guarantee both good performance and high resource efficiency during the system scaling phases, we should adapt soft resources fast and appropriately in the system, which determines the maximum request processing concurrency flowing to related servers.

3.3 Impact of Monitoring Time Interval

Based on our empirical studies, the accuracy of our SCT model highly depends on the main sequence curve extraction from the correlations between throughput and concurrency (see Fig. 6a), and such process would be affected by many factors in practice. The monitoring time interval of the runtime metrics measurement is one of the critical factors. For example, if the time interval is too long, we would not get enough points to estimate the range precisely and lose the ability to capture short-term congestion. On the other hand, if we set a too short monitoring time interval, the main sequence curve extraction can be negatively affected due to large variations for throughput and concurrency measurement. Concretely, the proper monitoring



(a) The comparison of the percentage of valid data when choosing different monitoring time intervals.

(b) The comparison of coefficiency of variation when choosing different monitoring time intervals.

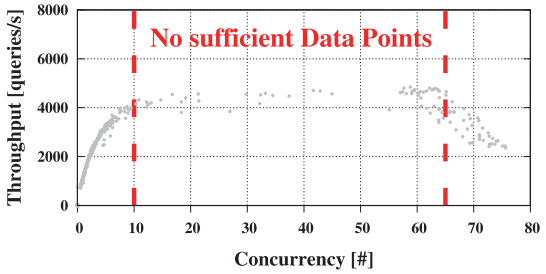
Fig. 7. Comparison of the average percentage of valid data and coefficiency of variation when we set different monitoring intervals. The proper monitoring time interval should guarantee a high percentage of valid data (PVD) and a low coefficiency of variation (CV). Under the current workload, we set the monitoring interval to be 50ms.

time interval should provide sufficient valid data and ensure minimized variation among data samples.

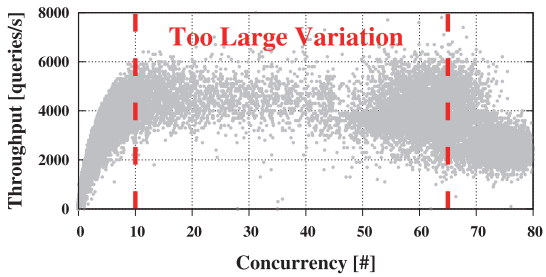
We formulate the impact of monitoring time interval on main sequence curve extraction as a function with two statistical metrics: *Percentage of Valid Data (PVD)* and *Coefficiency of Variation (CV)*. First, PVD is designed to identify whether a scatter graph has sufficient data for main sequence curve extraction. For each specific concurrency $Q_n (Q_n \in [Q_{min}, Q_{max}])$, we validate the throughput samples follows the normal distribution (i.e., $TP_{Q_n} \sim N(\mu, \sigma^2)$) through Kolmogorov-Smirnov test [32]. To mitigate the impact of data samples variation, we only extract the valid data samples which fall in the 95% confidence interval: $[\mu - 2\sigma, \mu + 2\sigma]$ and then the percentage of valid data for current concurrency should be a ratio of the number of valid data and number of data, denote as $PVD_{Q_n} = N_{PVD}/N$. Fig. 7a shows the comparison of the average PVD when we choose different monitoring intervals, we notice that we cannot set a too long time interval since we would not get sufficient valid data to arrange the estimation. After we set the time interval longer than 50ms, the percentage of valid data starts to decrease; for example, we only have 49% valid data when we set 1s time interval. Fig. 8a visualizes the scatter graph when we set the monitoring time interval to be 1s. Compared with Fig. 6a, insufficient data points make our model lose the ability to capture the short-term congestion of a server and our algorithm cannot estimate the optimal concurrency setting.

Second, we use coefficiency of variation (i.e., relative standard deviation) to evaluate the dispersion of throughput samples distribution, defined as $CV_{Q_n} = \overline{TP}/\sigma_{TP}$. This is because large variations for throughput measurement would degrade the main sequence curve extraction. For example, if we count the number of completed requests within a short time interval (e.g., 10ms), the requests would last several consecutive time intervals and too few requests would complete in a small interval. Then the throughput measurement would present large variations for each certain concurrency level (see Fig. 8b) and further blur the shape of the expected main sequence curve to degrade the estimation algorithm accuracy. From Fig. 7b, we finally select 50ms as a proper time interval setting since it can both have high PVD ($\geq 90\%$) and guarantee low CV ($\leq$ than 10%) at the same time.

In summary, we should both consider the percentage of valid data and the coefficiency of variation to choose the proper monitoring time interval. However, we cannot set constant thresholds for the two metrics to judge whether the metrics reach optimal because such a proper time interval is workload-dependent. We will study an automatic way to choose a proper time interval in future research.



(a) The scatter graph shows the correlation between MySQL throughput and concurrency measured with 1s time interval.



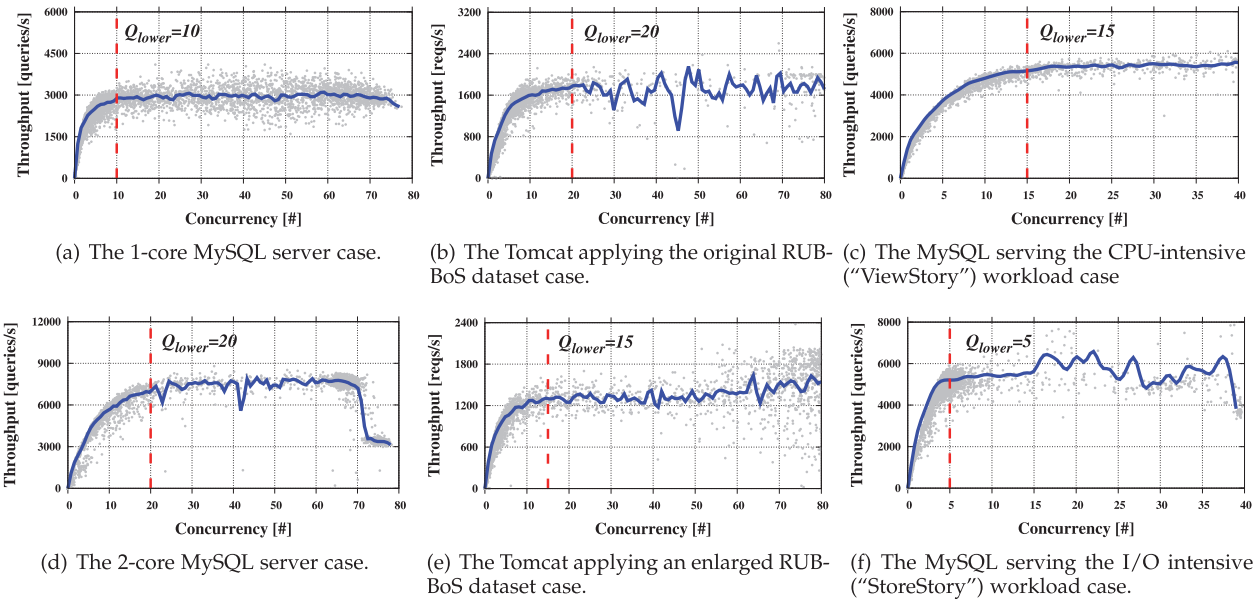
(b) The scatter graph shows the correlation between MySQL throughput and concurrency measured with 10ms time interval.

Fig. 8. The correlations between server concurrency and throughput when choosing too long a time interval (1s) and too short a time interval (10ms). Compared with Fig. 6b, a too long time interval would not provide sufficient valid data points to arrange the estimation, while a too short time interval would blur the shape of the expected main sequence curve due to the increased variation of throughput.

3.4 Factors That Affect Optimal Concurrency Setting

In this section, we use our SCT model to explore the relationship between optimal concurrency setting and runtime environment conditions. We study three common factors that could affect optimal concurrency setting in a web system scaling scenario: different hardware scaling strategies, system state change, and external workload type change. Our SCT model can capture the rapid shifts of the optimal concurrency without extensive profiling experiments required by offline performance model approaches.

1) Critical Resource Scaling Strategy: Vertical Scaling and Horizontal Scaling. Recall our experimental results in Section 2.2. We introduce that vertical scaling may incur the Tomcat server's optimal concurrency change through a set of profiling experiments. Unlike the offline profiling approach, the SCT model only requires collecting runtime metrics under the normal workload. For example, we configure our system with 1/4/1 hardware topology to serve browse-only CPU-intensive workload as in production, which causes MySQL to become the single bottleneck server in the system. We initially allocate a 1-core CPU for MySQL and gather the application-level metrics (sampled at 50ms granularity) for



(a) The 1-core MySQL server case. (b) The Tomcat applying the original RUBBoS dataset case. (c) The MySQL serving the CPU-intensive ("ViewStory") workload case

(d) The 2-core MySQL server case. (e) The Tomcat applying an enlarged RUBBoS dataset case. (f) The MySQL serving the I/O intensive ("StoreStory") workload case.

Fig. 9. The throughput-concurrency scatter graphs when server with different CPU allocations, RUBBoS dataset size, and serving workload types. Figs. 9a and 9d show the impact of CPU scaling up (e.g., 1-core to 2-core). Figs. 9b and 9e show the impact of dataset size change. Fig. 9c and 9f show the impact of workload types change. These case studies show that SCT model can capture the optimal concurrency shifting as runtime environment condition changes.

estimating the optimal concurrency. Moreover, we scale up the CPU to be 2-core and further generate another correlation. Figs. 9a and 9d display the correlations of MySQL throughput and concurrency under the 1-core MySQL and 2-core MySQL scenarios, respectively. Our SCT model shows that the optimal concurrency Q_{lower} doubles from 10 to 20 after vertical scaling.

We also study how horizontal scaling (i.e., add/remove # of instances) affects optimal concurrency settings. We allocate a 1-core CPU for MySQL and configure the system to be 1/4/2 hardware topology, indicating that we add a new MySQL server into the system. We notice that such horizontal scaling would not cause the optimal concurrency shifting due to the unchanged request service rate of the individual servers. However, horizontal scaling may incur performance variation since the capacity for the bottleneck tier has been changed. Soft resource re-adaption is still necessary for mitigating response time.

2) System State Change: Enlarge/Shrink RUBBoS Dataset. We prove that system state change caused by dataset updates can change server optimal concurrency settings. This is because the system state change would affect the degree of computation and further cause the request service rate variation. For instance, the dataset of web applications always updates due to continuous data refresh and synchronization, then the request service rate would oscillate along with the dataset size. We configure our system with 1/1/4 hardware topology to make Tomcat the bottleneck server. We exploit the default and a manually enlarged RUBBoS dataset. Comparing Figs. 9b with 9e, we observe the change of optimal Tomcat concurrency setting drops from 20 to 15 after RUBBoS dataset size-changing via concurrency-throughput correlation.

3) Workload Type Change: From CPU-intensive to I/O intensive. We also study how the workload type change affects the optimal concurrency settings for component servers. We use the original RUBBoS workload generator to send CPU-intensive workload (e.g., "ViewStory") and I/O-intensive workload (e.g., "StoreStory") separately. We prepare our system with 1/4/1 hardware topology to make MySQL the single bottleneck in the system. However, the critical hardware resource could change from CPU to disk I/O. Therefore, the server's capacity may change significantly due to such workload type change. Figs. 9c and 9f present the concurrency-throughput correlation of MySQL experiencing the CPU-intensive workload and the I/O-intensive workload, respectively. We observe that the optimal concurrency Q_{lower} drops from 15 to 5 after we change the "ViewStory" workload to the "StoreStory" workload.

These phenomena validate that the online SCT model can capture the near-optimal concurrency based on scatter graphs under various runtime environments, which builds the foundation to arrange dynamic soft resource reallocation for component servers in a web system.

4 Concurrency-Aware System Scaling Design and Implementation

We have introduced our SCT model that can quickly recommend the optimal concurrency settings of component servers in a web system on the fly. These experimental analyses prove the hardware-only scaling mechanisms cannot maintain the stable response time due to the mismatch between static soft resources allocation and shifting optimal concurrency setting along with the runtime environment changes (see Fig. 1). Hence, a fast rational adaption of soft resources becomes the key point to realize good performance and maintain high resource efficiency in an auto-scaling scenario. In this section, we implement the Concurrency-aware system Scaling (ConScale) framework, which combines the SCT model (see Section 3) with a hardware resource scaling mechanism to fast arrange an optimal resource adaption in

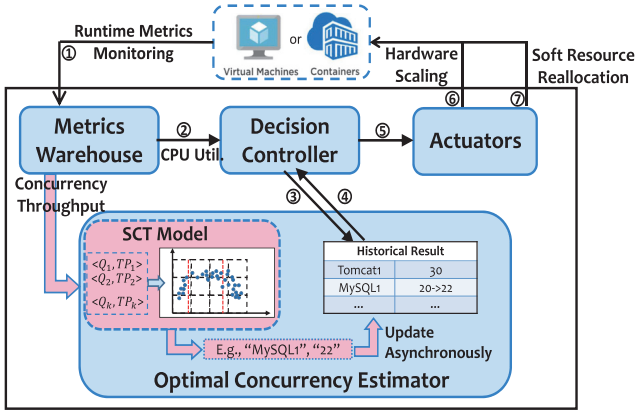


Fig. 10. The architecture of our ConScale framework.

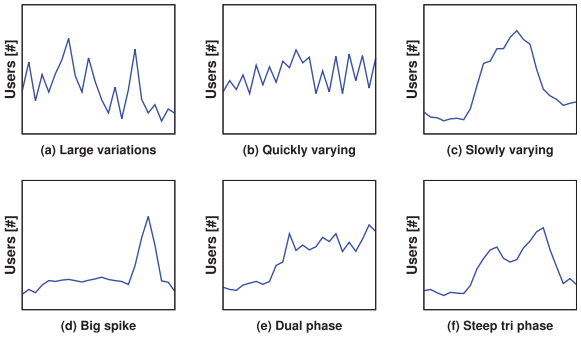


Fig. 11. Realistic workload traces used in our experiments.

the system during runtime. Fig. 10 introduces four components in our ConScale framework: Metric Warehouse, Online Optimal Concurrency Estimator, Decision Controller, and Actuators.

4.1 System Design

Metric Warehouse gathers system- and application-level metrics (e.g., CPU utilization and throughput) through the monitoring agents within each VM or container at every one second (step 1 in Fig. 10). We use Kafka [33] to serve as intermediate storage to coordinate the distributed monitor agents and generate available data for optimal concurrency estimation and threshold checking. *Decision Controller* decides the timing and actions to turn on/off VMs or containers in accordance with the system need and retrieves the soft resource allocation recommended from the Optimal Concurrency Estimator. It extracts threshold-related metrics from Metric Warehouse with a fixed time interval to decide whether initiate hardware scaling and soft resource reallocation. *Actuators* execute VM-scaling (or container-scaling) and soft resource reallocation decided by Decision Controller. *Optimal Concurrency Estimator* generates the optimal concurrency setting of key servers by continuously pulling metrics required by the SCT model (e.g., concurrency and throughput) from the metric warehouse asynchronously. Furthermore, it continuously updates the historical result table after a new optimal setting is generated based on the current workload. We set three minutes for building a new SCT model in our experiments. It is a reasonable time interval as it is long enough to gather sufficient metrics while short enough to adapt to runtime dynamics coordinating with autoscaling.

4.2 Implementations

VM-Scaling. Considering that the underlying hypervisor in clouds already provides sufficient APIs to manipulate VMs, we can easily manage the VMs by remotely calling these APIs. Nevertheless, we still mainly have two problems for VM scaling. First, scaling out stateful servers (e.g., database server) is tough work owing to the complex data consistency problem [34]. We can solve this problem by replicating the entire database when MySQL scales out since RUBBoS has a small dataset (i.e., 150MB archived). In the production environment, industry practitioners tend to set a long preparation period before launching each new VM. We set such a preparation period to be 15s after reasonable profiling tests.

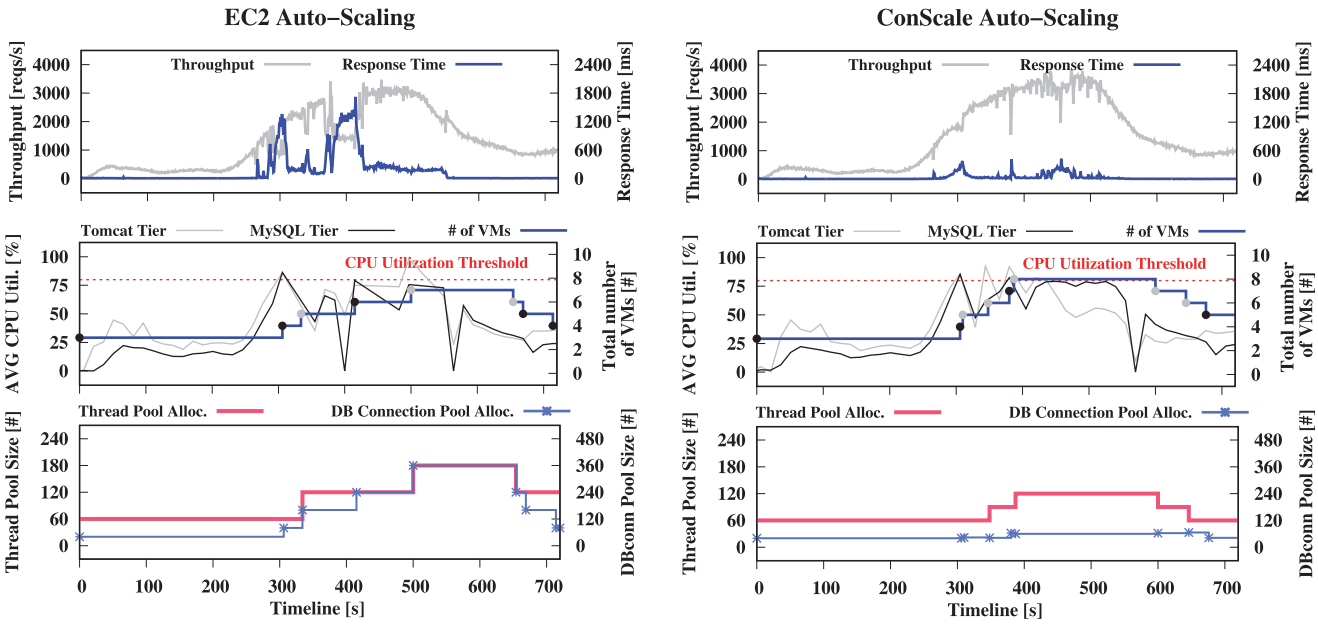
Second, we should deal with the load balancing problem among the existing servers in the system with the newly added servers after scaling. We use HAProxy [21] as a load balancer for managing the traffic to the application tier and the database tier since it supports both HTTP and TCP proxying. In other words, HAProxy would dispatch the requests from the upstream tier (e.g., web tier or app tier) to the downstream tier following pre-defined rules (i.e., load balancing policies). We adopt `leastconn` policy for both app tier and database tier balancing.

Container-Scaling. Our ConScale framework also applies to web applications deployed in a container-based virtualization system. Unlike the solution to data consistency and load balancing when we arrange VM-scaling, we choose to work with deployed container orchestration (e.g., Kubernetes[35]) to realize runtime container management such as load balancing and horizontal scaling. Load balancing distributes and loads among multiple container instances according to custom policy (e.g., `leastconn`). Horizontal scaling allows to add and remove containers during runtime with a very short period compared with launching VMs.

Soft Resource Re-Adaption. The Decision Controller automatically reallocates soft resources to control the maximum workload concurrency level of each server. In our implementation, we select two soft resource allocations: the thread pool size and DB connection pool size, which can limit the workload concurrency within the Tomcat server and MySQL server, respectively. Specifically, the latest Tomcat [36] supports Java Management Extensions (JMX [37]) technology to adapt the Tomcat thread pool size on the fly. Our software agent can arrange the thread pool size reconfiguration via RMI (Remote Method Invocation). Moreover, we slightly modified the RUBBoS source code to expose the interfaces of the DB connection pool size management and then applied RMI to realize runtime re-configuration for the database connection pool managed by JDBC.

5 Experiment Evaluation

We evaluate the strengths of our ConScale framework by comparing it to three auto-scaling solutions under the realistic bursty workload. To illustrate the applicability and effectiveness of our ConScale framework, we prepared two



(a) The large system response time spikes and throughput drops appear at 306s and 415s when system serving the "Slowly Varing" workload .

(b) The system maintains stable response time serving the "Slowly Varing" workload and the system throughput matches the workload variations as shown in Figure 11(a).

Fig. 12. System applying EC2-AutoScaling presents large response time spikes during the scaling phase when serving the "Slowly Varying" workload. The left three figures refer to the EC2-AutoScaling case and the right three figures summarize the ConScale case. ConScale outperforms EC2-AutoScaling and it helps the system maintain a stable response time since it limits the server concurrency to a rational level via soft resource reallocation.

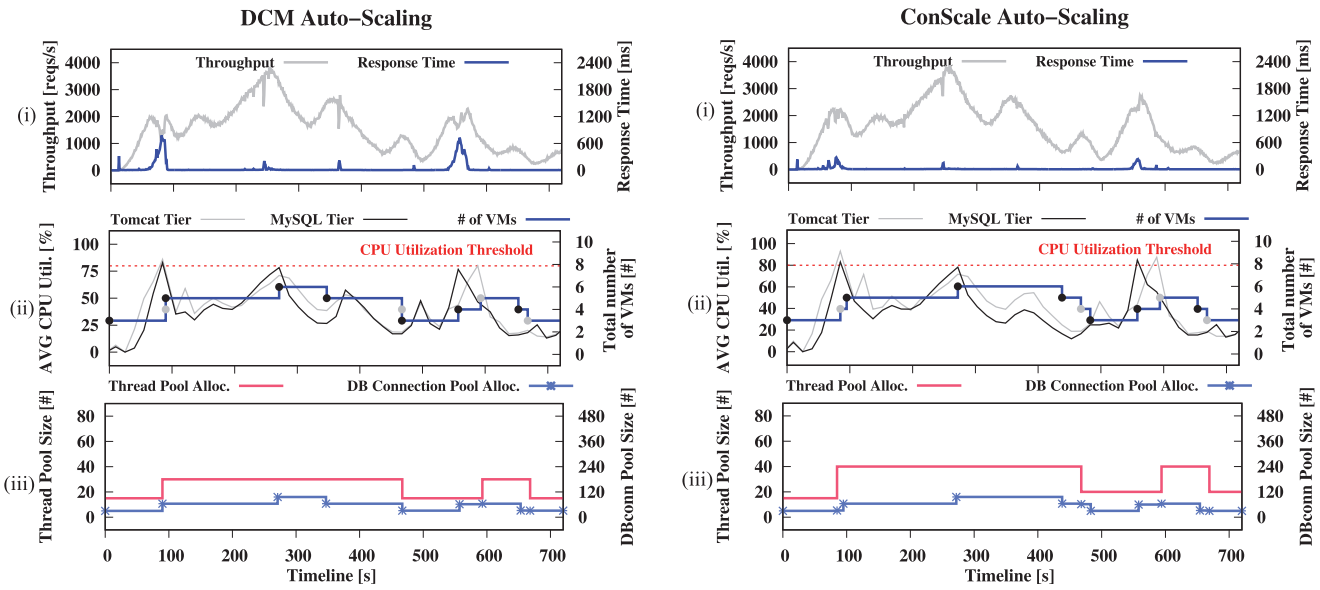
virtualization environments (i.e., VM-based and container-based) in our private cluster and set up two widely used hardware-only autoscalers for comparison: Amazon EC2-AutoScaling [8] and Kubernetes [35] default scaling solution: Horizontal Pod Autoscaler (HPA) [38]. We also compare ConScale with a state-of-the-art concurrency adapting management (i.e., DCM framework enabled concurrency-aware model [11]). ConScale outperforms the above three auto-scaling solutions in mitigating the response time fluctuations and producing high throughput thanks to the runtime optimal concurrency setting adaption recommended by our online SCT model.

5.1 Evaluation in VM-Based Virtualization System

Experimental Setup. Three auto-scaling frameworks (i.e., EC2-AutoScaling, DCM, and ConScale) are deployed in our private VMware ESXi cluster. EC2-AutoScaling monitors the usage of hardware resources (e.g., CPU or memory) by monitoring tools (e.g., Amazon CloudWatch [39]). Once the monitored target resource usage exceeds the pre-defined threshold (e.g., average CPU utilization $>$ 80%), the controller within such framework automatically adds/removes VMs to the bottleneck tier in the system. The DCM framework integrates dynamic soft resource reallocation according to the concurrency-aware model, which decides the soft resources via offline profiling before the production phase. We adopt the "quick start but slow turn off" strategy to avoid performance instability problem [9] and set a 15-second control period to handle data/state consistency problems caused by stateful server scaling. We evaluate three auto-scaling frameworks under six representative realistic workload traces (see Fig. 11). These workload traces are derived from real-world traces by Gandhi [9]. We continuously conduct a 12-minute experiment with 7500 maximum concurrent users.

*Performance Comparison Between ConScale and EC2-AutoScaling.*Fig. 12 presents the performance comparison between EC2-AutoScaling and ConScale frameworks when system serving the same "Slowly Varing" workload (see Fig. 11). The left three figures (Fig. 12a (i), (ii), (iii)) belong to the EC2-AutoScaling case, and the right three figures (Fig. 12b (i), (ii), (iii)) describe the ConScale case. We initialize our system from 1/1/1 hardware topology and 1000-60-40 soft resources allocation.

The system applying both EC2-AutoScaling and ConScale scales out key servers once the average CPU utilization of an individual tier exceeds a threshold (i.e., 80%). From Figs. 12a (i) and 12b(i), our ConScale helps system realize stable response time and throughput during the 12-min experiment period than that in the EC2-AutoScaling case. For instance, throughput drops and large response time spikes at the scaling out phases (e.g., periods 300s∼420s in Fig. 12a(i)) can be distinctly found in the EC2-AutoScaling case. Taking the period 300s∼320s in Fig. 12a(i) for example, Fig. 12a(ii) displays that MySQL scales out at 306s since the average CPU utilization of the original MySQL tier exceeds the threshold. Once the newly-added MySQL gets ready to serve incoming requests, the bottleneck tier shifts to Tomcat Tier, and the MySQL tier requires doubled concurrent requests since the total database connection pool size increases to 80. However, the total thread pool size in the upstream Tomcat restricts only 60 concurrent requests (see in Fig. 12a(iii)), leading to low efficiency of MySQL and further incurring the response time spike. The response time drops because new hardware resources are added after the control period to deal with the



(a) The system response time presents large response time spikes at timeline 56s and 556s serving "Large Variation" workload due to sub-optimal thread pool allocation in Tomcat.

(b) The system serving "Large Variation" workload with ConScale only present stable response time due to timely soft resource (thread pool in Tomcat) reallocation.

Fig. 13. ConScale framework supports the system maintains the optimal after the runtime environment condition changes. Comparing to the training process, we manually shrink the default RUBBoS dataset and the system presents a sub-optimal performance in the DCM case since the offline model cannot detect such variation.

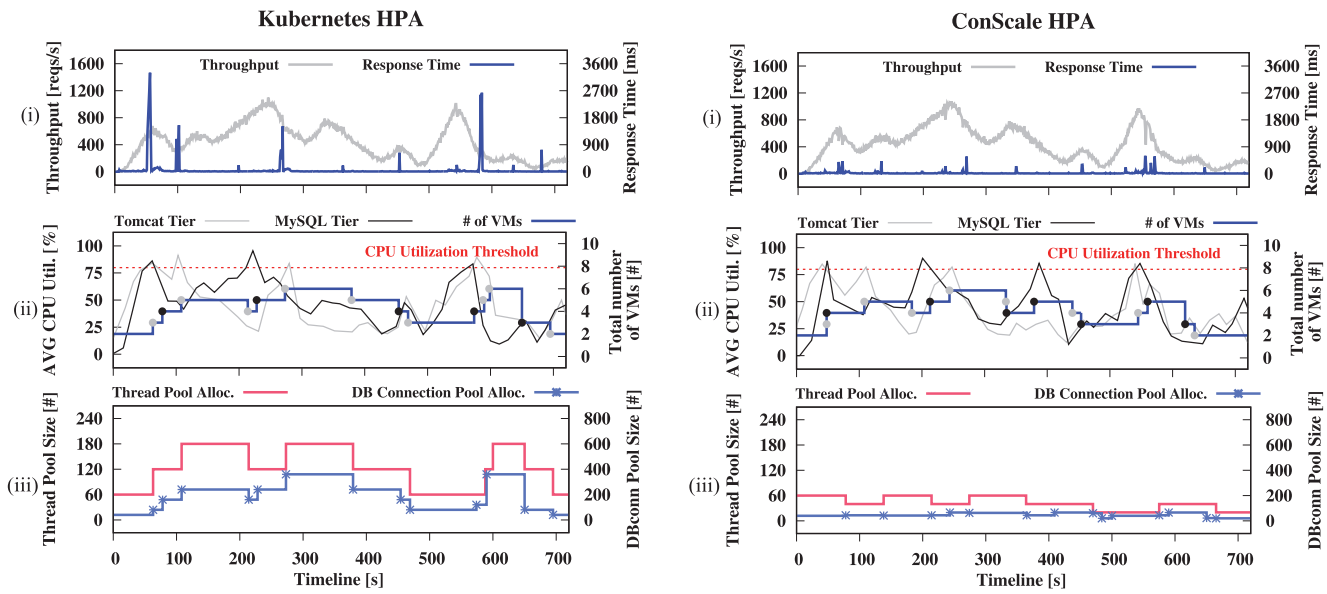
system overload. On the other hand, Fig. 12a(i) presents moderate response time fluctuations along with the system scaling phase. This is because our ConScale adapts the optimal soft resource recommended by the SCT model after hardware resource scaling, which can guarantee high efficiency of component server's hardware resources and realize more stable performance than that in the EC2-AutoScaling case. Beyond the scaling phases, ConScale achieves lower response time since the soft resource reallocation resolves the mismatch of soft and hardware resources. Compared with Figs. 12a(iii) and 12b(iii), ConScale can maintain the server concurrency limit at a rational level all the time via soft resource reallocation.

Performance Comparison between ConScale and DCM. We further validate the reliability and adaptiveness of our online SCT model by comparing the performance between ConScale and DCM, which uses the offline Concurrency-Aware model based on queueing network model [11]. The offline model requires a necessary training process to determine optimal soft resource allocation of key servers in the system under a specific workload. However, once the runtime system condition (e.g., critical resource, system state, and workload type) changes, DCM applying the previous "optimal" concurrency setting may lead to sub-optimal performance compared with our ConScale framework.

We work on the "Large Variation" trace and CPU-intensive workload. For DCM training, we configure an enlarged RUBBoS dataset in MySQL, and it recommends 15 for the thread pool size and 30 for the DB conn pool size in Tomcat. However, we manually reduce the dataset size to simulate the dataset updates in the production system. ConScale captures such environment condition change and estimates the new optimal concurrency setting of Tomcat shifts to 20, which is consistent with the analysis in Section 3.4. Fig. 13 shows that the DCM framework only achieves sub-optimal performance since the soft resource allocation in the system does not fit in the updated environment condition. For example, DCM generated an "optimal" concurrency setting for Tomcat should be 15 based on the previous system state. However, according to the recommendation of the SCT model, we should allocate 20 Tomcat threads for the new environment. Previous thread pool size setting becomes too low (comparing with Figs. 13a(iii) and 13b(iii)) and it may incur the under-allocation effect [18] on Tomcat server, i.e., low utilization of the hardware resources (e.g., CPU). Furthermore, the low efficiency of the Tomcat CPU impacts the system performance as shown in Fig. 13a(i). On the other hand, stable response time and controlled concurrency level in Fig. 13b(i) validate the accuracy and effectiveness of our online SCT model.

5.2 Evaluation in Container-Based Virtualization System

Experimental Setup. We implement and configure Kubernetes on eight two-core VMs in our private VMware ESXi cluster. Kubernetes is a commonly used lightweight open-source container management platform that can orchestrate containers and provide Horizontal Pod AutoScaler (HPA) to add or remove containers to handle the bursty workload. Kubernetes defines that a pod can be a group of containers that are tightly coupled together with a shared IP address and port space [40] We simply regard one pod refers to a single container. Kubernetes HPA exploits a control loop algorithm based on CPU utilization reported by each Pod to determine the correct number of pods to keep the average CPU utilization at a target value (e.g., 80%). On the other hand, ConScale collects each pod CPU usage and adapts



(a) The system response time presents large response time spikes under the "Large Variation" workload at timeline 58s, 102s, 268s, and 580s due to liberal soft resource allocation.

(b) The system response time is stable and low under the same workload in (a) since ConScale HPA limit soft resource allocation both for Tomcat and MySQL service.

Fig. 14. Our ConScale framework can maintain a more stable response time compared to the Kubernetes HPA case. The performance degradation caused by liberal soft resource allocation still exists when we shift to deploy the service using containers. Our approach can also be applied to each individual Pod/Container of a microservice system.

Decision Controller to adopt the same algorithm as Kubernetes HPA to decide when to adjust the number of pods and further arranges soft resource allocation after pod scaling.

We conduct the evaluation experiments to compare the performance when the system applies Kubernetes HPA and our ConScale framework under six realistic workload traces in Fig. 11. We decreased the maximum concurrent users during 12 minutes to be 1000 to adapt to the Kubernetes Pod with 2-core vCPU allocation.

*Performance Comparison Between ConScale and HPA.*We further adapt our ConScale to container-based cloud systems (e.g., Microservice systems). Microservice systems always exploit lightweight container-based virtualization in comparison to Virtual Machines(VMs) since the container can be instantiated, terminated, and managed very quickly, which provides better auto-scaling improvements on application response time [41]. Our approach can be applied to each individual container of a microservice system based on runtime metrics from the container.

Here we compare the performance between Kubernetes HPA and ConScale in the container-based virtualization system. Comparing Figs. 14a(i) with 14b(i), ConScale presents relatively stable response time during the whole experiment period in contrast to Kubernetes HPA case. We found that the response time spikes in Fig. 14a(i) match the scaling out actions of Tomcat and MySQL Pods. Similar to the EC2-AutoScaling case, even though the Kubernetes HPA adopts an algorithm to calculate the correct number of pods to workload, adding or removing component pods can implicitly change the concurrency of upstream and downstream pods thus impacting the overall performance. Fig. 14b(ii), (iii) show the server concurrency would not be limited at a rational level via default soft resource arrangement for the whole experiment period. Our ConScale adjusts the soft resource allocation for individual pod according to runtime optimal concurrency setting estimation made by our SCT model. From Fig. 14b(i), (ii), and (iii), we found that during the scaling out phase, the large response time spikes are mitigated after we apply the optimal concurrency setting.

Table 1 summarizes the tail response time (i.e., 95th and 99th percentile) comparison between EC2-AutoScaling, DCM, Kubernetes, and ConScale cases under six categories of workload traces (see Fig. 11). Our results demonstrate that ConScale can restrict the 95th and 99th response time below 500ms (the requirement for most web applications [42]) when the system serves all categories of traces.

5.3 Quantitaive Analysis of Overhead of ConScale Framework

ConScale collects application-level metrics for optimal concurrency setting estimation based on each component server's request processing log, which records the arrival and the departure timestamps of individual requests within that server at millisecond granularity. We utilize each server's default logging function to avoid causing additional overhead to the server. Based on our current experimental setup, we have two servers to be monitored (e.g., Tomcat and MySQL). For the Tomcat server, we directly extract the request processing timestamps from *localhost_access.log*. For the MySQL server, we record the query timestamps when each query is sent from Tomcat to MySQL and when Tomcat receives the corresponding response from MySQL. To reduce the overhead caused by the amount of data, we only note the arrival and departure timestamps of individual requests in Tomcat.

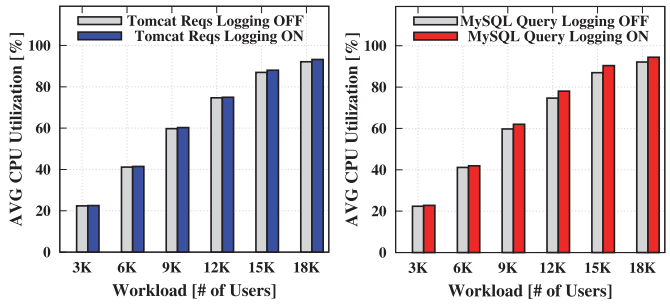
We conduct experiments to quantitatively evaluate the overhead caused by such request/query logging. We set up the system in 1/1/4 hardware topology to make Tomcat the bottleneck server. At each workload, we ran experiments for three minutes and measure the average Tomcat CPU utilization. Fig. 15a shows the comparison of average CPU

TABLE 1
Comparison of Tail Latency (I.E., 95Th and 99Th Percentile Response Time) When the System Applies EC2-AutoScaling, DCM, Kubernetes HPA and ConScale to Serve Six Categories Workload Traces, Respectively

	Percentile Response time [ms]		Large *Variation*	*Quick Varying*	*Slowly Varying*	*Big Spike*	*Dual Phase*	*Steep Tri Phase*
VM-based AutoScaling	RT_{95th}	EC2-AutoScaJing	462	157	1135	687	225	101
		DCM	274	92	157	367	192	75
		ConScale	**157**	**48**	**85**	**179**	**81**	**56**
	RT_{99th}	EC2-AutoScaJing	2345	684	3252	3981	1153	1259
		DCM	1080	443	1499	1376	606	537
		ConScale	**465**	**229**	**218**	**479**	**328**	**171**
Container-based AutoScaling	RT_{95th}	Kubernetes-HPA	213	120	276	252	121	164
		ConScale	**79**	**36**	**56**	**60**	**85**	**49**
	RT_{99th}	Kubernetes-HPA	1004	707	1534	1103	962	876
		ConScale	**266**	**121**	**276**	**246**	**187**	**159**

utilization of the Tomcat server with logging turned on/off at each workload. We can see that the overhead is always at a low level, reaching the maximum of 1.1% at workload 18000. On the other hand, Fig. 15b shows that logging MySQL queries cause more overhead than logging Tomcat requests since the number of queries is more than requests (one client request to Tomcat may trigger multiple queries to MySQL). Nevertheless, this figure shows a maximum of 3.4% overhead at workload 15000, which is still at a low level for the whole system. Such quantitative overhead analysis demonstrates the overall low overhead of our ConScale framework.

We simply analyze why our ConScale framework maintains such a low-level overhead on request logging. First of all, the logging tools we used (Tomcat Access logging and Apache Log4j) are mature in respect of avoiding overhead. The Access Log Valve uses self-contained logic to write its log files[43] and Log4j uses asynchronous logging to guarantee high throughput[44]. Second, we take the initiative to avoid overhead by regulating the pattern for request/query logging. We only record the arrival and departure timestamps, which helps control the amount of data. Third, we guarantee the hard disk I/O not be the bottleneck. Even the CPU cycles occupied by logging increase at peak workload, the majority of CPU cycles are still utilized by service, which accounts for the limited CPU utilization increasing in Fig. 15. Hence, our ConScale framework can maintain a low-level logging overhead and can be easily integrated into existing system scaling mechanisms.



(a) The comparison of average CPU utilization of Tomcat server when ConScale turns on/off Tomcat requests logging at each workload.

(b) The comparison of average CPU utilization of Tomcat server when ConScale turns on/off MySQL queries logging at each workload.

Fig. 15. Quantitative analysis of overhead of recording request/query processing log. We conduct experiments to measure the average CPU utilization of Tomcat when ConScale turns on/off recording request/query processing log at each workload. Our framework only causes a maximum 4.82% overhead at peak workload.

6 Related Work

Rule-Based Auto-Scaling Mechanism is commonly used for computing resource management in clouds among industry and academia. The benefit of such rule-based auto-scaling is the convenience for deploying, however, how to decide such a threshold becomes an obstacle to maintain good performance under bursty workloads. For VM-based autoscalers, Dutreilh *et al.* [45] proposes that such auto-scaling mechanism should consider the reaction of the system to scaling actions by setting up cool-down or inertia period. Hasan *et al.* [46] set a detailed threshold for scaling decision that depends on multiple resources (e.g., CPU, memory, or network) correlation. On the other hand, many approaches for scaling containers also contribute to threshold design. Horovitz *et al.* [17] use reinforcement learning to adapt the scaling thresholds on the fly. Gotin *et al.* [47] investigate which performance metrics to be used by a threshold-based autoscaler. Our work similarly correlates hybrid metrics consisting of average CPU usage of each tier and application-level metrics from the key server (e.g., concurrency and throughput) to compose a robust and reliable threshold.

Auto-Scaling Mechanism With Resource Adaption. Past studies [48], [49] illustrate that the multiple dependencies among individual tiers in an n-tier system may impact system performance in scaling scenario. Many past studies contribute to the study that integrating critical resource adaption with common system scaling can handle such problems in clouds. Nathuji *et al.* [48] implement Q-clouds, which can be aware of

QoS of applications and adjust hardware resource allocations to handle performance interference effects. Kalyvianaki *et al.* [50] integrates the Kalman filter with feedback controllers to detect and adapt to unpredictable workload changes via CPU resource reallocation of each VM in the cluster. Sun *et al.* [49] develop ROAR, which uses model-based analysis and load test to generate optimal cloud resource allocations to meet the QoS requirement. Our work contributes to generating updated soft resources configuration for optimal performance during runtime and integrates such resource adaption management with the existing auto-scaling schema.

Software Reconfiguration to optimize system performance has been explored extensively before. For example, Sriraman *et al.* [4] develop μTune to select load-optimal threading models for the application server to mitigate SLO violations. Gunther *et al.* [51] characterize the relationship between threads concurrency and server performance in a single server environment. Zhang *et al.* [52] build machine learning models to automatically generate optimal database configurations for improving throughput. Maji *et al.* [53] study the impact of parameters (e.g., MaxClients and KeepaliveTimeout) inside Apache web server on overall performance in a shared cloud environment. Our work studies the impact of soft resource allocation on hardware resource efficiency rather than parameter tuning, which complements their work by using a runtime correlation model to tune soft resource allocation in each individual server of an n-tier system.

7 Conclusion

We studied that the online identification of the optimal concurrency for individual servers in a web system contributes to dynamically adapting soft resources to maintain stable response time along with hardware resource variation in the auto-scaling scenario. Based on experiments using the RUBBoS n-tier benchmark, we reveal that several factors lead to the shifting of optimal concurrency to reach the highest throughput (Sections 2.2 and 3.4). We then develop a Scatter-Concurrency-Throughput (SCT) model to determine the optimal concurrency of a server in a web system during runtime via fine-grained application-level metrics (Section 3). Furthermore, we implement a Concurrency-aware system Scaling (ConScale) framework which can fast and intelligently adapt optimal soft resource of key servers recommended by the SCT model along with hardware resource scaling (Section 4). Our evaluation experiments compared to state-of-the-art system scaling managements show that ConScale can help various large-scale systems effectively maintain a stable response time and guarantee low long-tail latency (Section 5).

Acknowledgments

Any opinions, findings, and conclusions are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or other funding agencies and companies mentioned above.

References

[1] C. Qu, R. N. Calheiros, and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey," *ACM Comput. Surv.*, vol. 51, no. 4, 2018, Art. no. 73.

[2] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson, "Characterizing, modeling, and generating workload spikes for stateful services," in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 241–252.

[3] B. Snyder, "Server virtualization has stalled, despite the hype," *Despite Hype*, Dec. 2010. [Online]. Available: https://www.infoworld.com/article/2624771/

[4] A. Sriraman and T. F. Wenisch, "tune: Auto-tuned threading for OLDI microservices," in *Proc. 13th USENIX Symp. Oper. Syst. Des. Implementation*, 2018, pp. 177–194.

[5] B. Vamanan, J. Hasan, and T. Vijaykumar, "Deadline-aware datacenter TCP (D2TCP)," in *Proc. ACM SIGCOMM Conf. Appl., Technol., Archit. Protoc. Comput. Commun. Rev.*, 2012, pp. 115–126.

[6] R. Rojas-Cessa , Y. Kaymak, and Z. Dong, "Schemes for fast transmission of flows in data center networks," *IEEE Commun. Surv. Tut.*, vol. 17, no. 3, pp. 1391–1422, Jul.–Sep. 2015.

[7] S. Adler, "The slashdot effect: An analysis of three internet publications," *Linux Gazette*, vol. 38, no. 2, p. 623, 1999.

[8] A. W. Services, "Amazon EC2 auto scaling," Aug. 2019. [Online]. Available: https://aws.amazon.com/ec2/autoscaling/

[9] A. Gandhi, M. Harchol-Balter , R. Raghunathan, and M. A. Kozuch, "Autoscale: Dynamic, robust capacity management for multi-tier data centers," *ACM Trans. Comput. Syst.*, vol. 30, no. 4, pp. 1–26, 2012.

[10] R. Han, L. Guo, M. M. Ghanem, and Y. Guo, "Lightweight resource scaling for cloud applications," in *Proc. 12th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput.*, 2012, pp. 644–651.

[11] Q. Wang, H. Chen, S. Zhang, L. Hu, and B. Palanisamy, "Integrating concurrency control in n-tier application scaling management in the cloud," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 4, pp. 855–869, Apr. 2019.

[12] Q. Wang *et al.*, "Optimizing N-tier application scalability in the cloud: A study of soft resource allocation," *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 4, no. 2, pp. 1–27, 2019.

[13] T. Lorido-Botrán, J. Miguel-Alonso , and J. A. Lozano, "Auto-scaling techniques for elastic applications in cloud environments," University of Basque Country, Leioa, Biscay, Spain, Tech. Rep. EHU-KAT-IK-09, 2012.

[14] Q. Zhang, L. Liu, C. Pu, Q. Dou, L. Wu, and W. Zhou, "A comparative study of containers and virtual machines in big data environment," in *Proc. IEEE 11th Int. Conf. Cloud Comput.*, 2018, pp. 178–185.

[15] T. Zheng *et al.*, "SmartVM: A SLA-aware microservice deployment framework," *World Wide Web*, vol. 22, no. 1, pp. 275–293, 2019.

[16] A. Bauer, V. Lesch, L. Versluis, A. Ilyushkin, N. Herbst, and S. Kounev, "Chamulteon: Coordinated auto-scaling of micro-services," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst.*, 2019, pp. 2015–2025.

[17] S. Horovitz and Y. Arian, "Efficient cloud auto-scaling with SLA objective using Q-learning," in *Proc. IEEE 6th Int. Conf. Future Internet Things Cloud*, 2018, pp. 85–92.

[18] Q. Wang *et al.*, "The impact of soft resource allocation on n-tier application scalability," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2011, pp. 1034–1045.

[19] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1984.

[20] O. Consortium, "RUBBoS: Bulletin board benchmark," Aug. 2017. [Online]. Available: http://jmob.ow2.org/rubbos.html

[21] H. Community, "HAProxy: The reliable, high performance TCP/HTTP load balancer," Aug. 2019. [Online]. Available: http://www.haproxy.org/

[22] Dormando, "Memcached," Aug. 2019. [Online]. Available: https://memcached.org/

[23] VMware, "VMware ESXI: The purpose-built bare metal hypervisor," Aug. 2019. [Online]. Available: https://www.vmware.com/products/esxi-and-esx.html

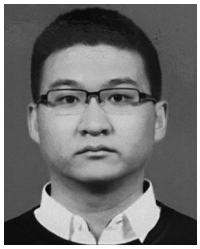
[24] T. Khare, *Apache Tomcat 7 Essentials*. Birmingham, U.K.: Packt Publishing Ltd., 2012.

[25] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, "Cloudscale: Elastic resource scaling for multi-tenant cloud systems," in *Proc. 2nd ACM Symp. Cloud Comput.*, 2011, pp. 1–14.

[26] Kubernetes vertical pod auto-scaling, Aug. 2019. [Online]. Available: https://cloud.google.com/kubernetes-engine/docs/concepts/verticalpodautoscaler

[27] P. Greenfield, D. Kuo, S. Nepal, and A. Fekete, "Consistency for web services applications," in *Proc. 31st Int. Conf. Very Large Data Bases*, 2005, pp. 1199–1203.

[28] S. Malkowski, M. Hedwig, J. Parekh, C. Pu, and A. Sahai, "Bottleneck detection using statistical intervention analysis," in *Proc. Int. Workshop Distrib. Syst.: Operations Manage.*, 2007, pp. 122–134.
[29] M. Welsh, D. Culler, and E. Brewer, "SEDA: An architecture for well-conditioned, scalable internet services," in *ACM SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, pp. 230–243, 2001.
[30] Y. Diao, J. L. Hellerstein, S. Parekh, H. Shaikh, and M. Surendra, "Controlling quality of service in multi-tier web applications," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst.*, 2006, pp. 25–25.
[31] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Amsterdam, The Netherlands: Elsevier, 2011.
[32] Z. Drezner, O. Turel, and D. Zerom, "A modified kolmogorov–smirnov test for normality," *Commun. Statist. - Simul. Comput.*, vol. 39, no. 4, pp. 693–704, 2010.
[33] A. S. Foundation, Apache kafka: A distributed streaming platform, 2019. [Online]. Available: https://kafka.apache.org/
[34] D. Abadi, "Consistency tradeoffs in modern distributed database system design: Cap is only part of the story," *Computer*, vol. 45, no. 2, pp. 37–42, 2012.
[35] Kubernetes, "Kubernetes," Jan. 2020. [Online]. Available: https://kubernetes.io/
[36] A. S. Foundation, Apache tomcat 7, 2019. [Online]. Available: https://tomcat.apache.org/tomcat-7.0-doc/funcspecs/mbean-names.html
[37] Oracle, "Java management extensions (JMX) technology," Aug. 2019. [Online]. Available: https://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html
[38] Kubernetes horizontal pod auto-scaling, Aug. 2019. [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
[39] A. W. Services, "Amazon cloudwatch: Complete visibility of your cloud resources and applications," Aug. 2019. [Online]. Available: https://aws.amazon.com/cloudwatch/
[40] S. Taherizadeh and M. Grobelnik, "Key influencing factors of the kubernetes auto-scaler for computing-intensive microservice-native cloud-based applications," *Adv. Eng. Softw.*, vol. 140, 2020, Art. no. 102734.
[41] N. Kratzke, "About microservices, containers and their underestimated impact on network performance," 2017, *arXiv:1710.04049*.
[42] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, 2013.
[43] Apache tomcat 7 the valve component, 2020. [Online]. Available: https://tomcat.apache.org/tomcat-7.0-doc/config/valve.html
[44] Apache, "Logging services," Jan. 2020. [Online]. Available: https://logging.apache.org/log4j/2.x/performance.html
[45] X. Dutreilh, A. Moreau, J. Malenfant, N. Rivierre, and I. Truck, "From data center resource allocation to control theory and back," in *Proc. IEEE 3rd Int. Conf. Cloud Comput.*, 2010, pp. 410–417.
[46] M. Z. Hasan, E. Magana, A. Clemm, L. Tucker, and S. L. D. Gudreddi, "Integrated and autonomic cloud resource scaling," in *Proc. IEEE Netw. Oper. Manage. Symp.*, 2012, pp. 1327–1334.
[47] M. Gotin, F. Lösch, R. Heinrich, and R. Reussner, "Investigating performance metrics for scaling microservices in cloudiot-environments," in *Proc. ACM/SPEC Int. Conf. Perform. Eng.*, 2018, pp. 157–167.
[48] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: Managing performance interference effects for QoS-aware clouds," in *Proc. 5th Eur. Conf. Comput. Syst.*, 2010, pp. 237–250.
[49] Y. Sun, J. White, S. Eade, and D. C. Schmidt, "ROAR: A QoS-oriented modeling framework for automated cloud resource allocation and optimization," *J. Syst. Softw.*, vol. 116, pp. 146–161, 2016.
[50] E. Kalyvianaki, T. Charalambous, and S. Hand, "Adaptive resource provisioning for virtualized servers using kalman filters," *ACM Trans. Auton. Adaptive Syst.*, vol. 9, no. 2, pp. 1–35, 2014.
[51] N. Gunther, S. Subramanyam, and S. Parvu, "A methodology for optimizing multithreaded system scalability on multi-cores," May 2011. [Online]. Available: http://arxiv.org/abs/1105.4301
[52] B. Zhang *et al.*, "A demonstration of the ottertune automatic database management system tuning service," in *Proc. VLDB Endowment*, 2018, pp. 1910–1913.
[53] A. K. Maji, S. Mitra, B. Zhou, S. Bagchi, and A. Verma, "Mitigating interference in cloud services by middleware reconfiguration," in *Proc. 15th Int. Middleware Conf.*, 2014, pp. 277–288.



Jianshu Liu (Member, IEEE) received the BEng degree from the Beijing University of Posts and Telecommunications, China, in 2018. He is currently working toward the PhD degree under the supervision of Dr. Qingyang Wang with the Department of EECS, Louisiana State University-Baton Rouge. His research interests include the performance and scalability analysis of monolithic and microservice-based architectures, with the aim of achieving highly responsive web applications running at high utilization in the cloud.

Shungeng Zhang (Member, IEEE) received the BEng degree from the Huazhong University of Science and Technology, China, in 2014, and the PhD degree in computer science from Louisiana State University–Baton Rouge in 2021. He is currently an assistant professor with the School of Cyber and Computer Sciences, Augusta University. His research interests include distributed systems and cloud computing, which focuses on improving the performance and scalability of large-scale web applications and IoT stream processing systems to simultaneously achieve both good performance and high resource utilization efficiency in the cloud.

Qingyang Wang (Member, IEEE) received the BSc degree from the Chinese Academy of Sciences in 2004, the MSc degree from Wuhan University in 2007, and the PhD degree in computer science from Georgia Tech in 2014. He is currently an associate professor with the Department of EECS, Louisiana State University-Baton Rouge. His research interests include distributed systems and cloud computing with a current focus on performance and scalability analysis of largescale web applications, such as, Amazon.com. He has led research projects with LSU on cloud performance measurements, scalable web application design, and automated system management in clouds.

Jinpeng Wei (Member, IEEE) received the BSc and MSc degrees from Wuhan University, and the PhD degree in computer science from Georgia Tech. He is currently an associate professor with the Department of Software and Information Systems, College of Computing and Informatics, University of North Carolina at Charlotte. His research interests include secure computer systems, including stealthy malware detection and defense and botnet C&C covert channels, applying systems virtualization to build high-performance applications, such as service-oriented computing architectures.