Mitigating Large Response Time Fluctuations through Fast Concurrency Adapting in Clouds

Jianshu Liu*, Shungeng Zhang*, Qingyang Wang*, Jinpeng Wei†

*Louisiana State University–Baton Rouge, †University of North Carolina–Charlotte

***Abstract*—Dynamically reallocating computing resources to handle bursty workloads is a common practice for web applications (e.g., e-commerce) in clouds. However, our empirical analysis on a standard n-tier benchmark application (RUBBoS) shows that simply scaling an n-tier application by reallocating hardware resources without fast adapting soft resources (e.g., server threads, connections) may lead to large response time fluctuations. This is because soft resources control the workload concurrency of component servers in the system: adding or removing hardware resources such as Virtual Machines (VMs) can implicitly change the workload concurrency of dependent servers, causing either under- or over-utilization of the critical hardware resource in the system. To quickly identify the optimal soft resource allocation of each server in the system and stabilize response time fluctuation, we propose a novel Scatter-Concurrency-Throughput (SCT) model based on the monitoring of each server's real-time concurrency and throughput. We then implement a Concurrency-aware system Scaling (ConScale) framework which integrates the SCT model to fast adapt the soft resource allocations of key servers during the system scaling process. Our experiments using six realistic bursty workload traces show that ConScale can effectively mitigate the response time fluctuations of the target web application compared to the state-of-the-art cloud scaling strategies such as EC2-AutoScaling.**

***Index Terms*—scalability, soft resources, web applications**

I. Introduction

Scalability is a key feature for modern cloud platforms: an application can dynamically add or remove its computing resources (e.g., VMs or CPU cores) to meet the requirement of varying workloads. For modern web-facing applications such as e-commerce, scalability is especially important since their workloads are naturally bursty. For example, customers visiting Amazon.com during holidays (e.g., Black Friday) could be 10X higher than that in other normal days [1]. Even within the same day, the peak workload during the rush hours can be significantly higher than the midnight traffic. The traditional practice of always provisioning the system for peak workloads will waste a significant amount of computing resources and power due to low resource utilization (e.g., averagely 18% [2]). Thus, the ability to dynamically scale a web application to match the real-time workload need is of critical importance.

Unlike embarrassingly parallel batch workloads such as MapReduce and Hadoop, effectively scaling a web application is especially challenging due to two reasons. First, most web applications have strict Quality of Service (QoS) requirements. For example, web search requires 99th percentile response time < 300ms [3]–[5]). Due to the bursty nature of web application workloads, it is challenging to intelligently scale the necessary computing resources to adapt to the dynamic workload variations and always meet the SLAs. Some previous research efforts have adopted a proactive approach to predict near-future workload [6], [7], however, predicting n-tier application workloads such as e-commerce is a well-known research challenge because of the bursty nature of the workload (e.g., Slashdot effect [8]). So temporary overloading of the system is unavoidable in practice. Figure 1 shows the large response time fluctuation of a 3-tier system when it scales the number of VMs using the EC2-AutoScaling strategy [1] to handle the workload variation. We frequently observed large response time spikes during the system scaling phase due to the temporary overloading.

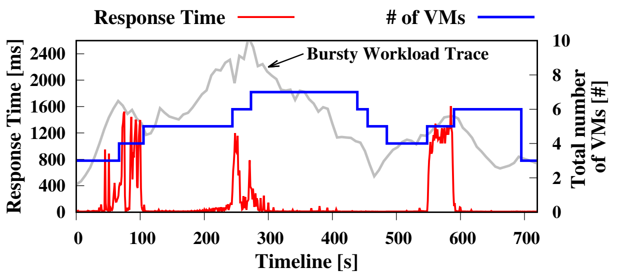


Fig. 1: Large response time fluctuations of a 3-tier system when it scales the number of VMs using the EC2-AutoScaling strategy to handle bursty workload.

Second, beyond hardware resources, many configuration issues such as soft resource allocations (e.g., server threads or connections) have a significant impact on web application performance. For example, the previous research [10] shows that scaling such as adding or removing VMs in an n-tier system also changes the workload concurrency of dependent servers, which may cause either under- or over-utilization of the critical hardware resource in the system. Thus hardware-only scaling solutions [6], [7], [11] such as EC2-AutoScaling may not gain the full potential of newly added hardware resources without adapting soft resource allocations. Even worse, the imbalance between hardware and soft resources would incur performance degradation instead of performance improvement [12]. More recent research [10] integrates the concurrency adaption (through soft resource reallocation) with the hardware resource scaling in their cloud system scaling management. However, their concurrency adaption for each

[1]EC2-AutoScaling applies a simple utilization threshold (e.g., 80%) on resources such as CPU to make scaling decisions. [9]

hardware scaling is based on static pre-profiling results before the production phase. Once the production runtime environment (e.g., the system state, workload characteristic, and scaling strategy) changes from the pre-profiling conditions, the static profiling approach could generate sub-optimal or even harmful recommendations of soft resource allocations.

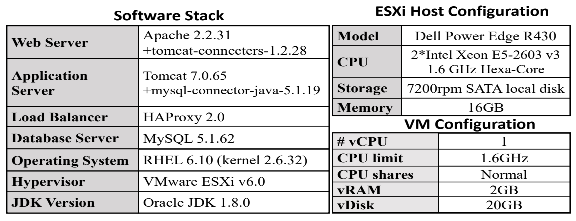
In this paper, we propose a novel online Scatter-Concurrency-Throughput (SCT) model which can quickly recommend the optimal soft resource allocations for each server based on the real-time monitoring of each server's concurrency and throughput. We assume each server in a web system keeps a request processing log, which records the arrival and departure time of each request of the server at millisecond granularity. For sufficiently short time intervals (e.g., 50ms), we can use the request completion rate as the server's real-time throughput, and concurrent requests as server's real-time concurrency. Based on the classic Utilization Law [13], the optimal concurrency of a server is the minimal concurrency that can achieve the highest server throughput. Given the bursty nature of n-tier application workload and continuous monitoring of each server's real-time concurrency and throughput, our approach can catch the workload concurrency variation and reveal the optimal concurrency of each server on the fly based on the correlation analysis of the two collected metrics.

We implement a Concurrency-aware system Scaling (ConScale) framework that integrates our SCT model with the hardware resource scaling into our system scaling management. ConScale exploits the SCT model to estimate an up-to-date rational concurrency setting of each server in the system by analyzing fine-grained measured data (concurrency and throughput) of each server. Specifically, ConScale takes two steps during a system scaling activity. First, scale-out/in hardware resources using the classic threshold-based hardware-only scaling strategy (e.g. EC2-AutoScaling). Second, once the hardware scaling is done, ConScale adjusts the soft resources of each dependent server based on the recommendations of the SCT model. By taking both hardware resources and the updated optimal soft resource allocations into consideration, our ConScale framework can stabilize the system response time during the system scaling process, especially during the temporary overloading phases.

In brief, we summarize our contributions as follows:

- Develop the online SCT model which can quickly determine the optimal soft resource allocation of each server in an n-tier application.
- Conduct an empirical study revealing several factors that would affect the optimal concurrency setting of different types of servers (e.g., Tomcat vs. MySQL).
- Implement the ConScale framework to realize fast and intelligent soft resources adaption to match the hardware resource variations in system scaling scenarios in clouds.

The rest of the paper is organized as follows. Section II shows our motivation experimental results that the optimal concurrency for a specific server is determined by various system factors (e.g., the system state, workload characteristic, and scaling strategy). Section III illustrates the online Scatter-Concurrency-Throughput model. Section IV introduces the design of our ConScale framework and implementation details. Section V displays the experimental evaluation under six categorized realistic workload. Section VI summarizes the related work, and Section VII concludes the paper.



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

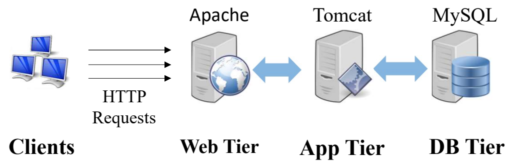
ESXi Host Configuration

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: Detailed experimental setup.

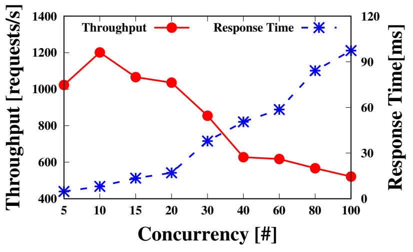
II. Background and Motivation

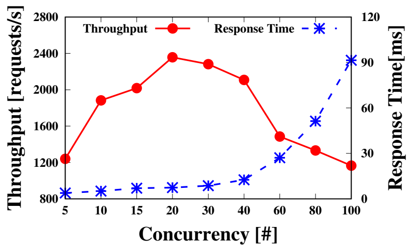
A. Experiment Setup

We adopt a standard n-tier benchmark RUBBoS [14], which is modeled after a bulletin board applications like Slashdot [8]. We deploy the RUBBoS benchmark as a three-tier system (web server tier, application server tier, and database server tier), more tiers can be configured on-demand (load balancer tier like HAProxy [15] or cache tier like Memcached [16]). There are 24 servlets providing different interactions such as "ViewStory", which can be categorized into two workload modes: browse-only CPU-intensive and read/write-mix I/O-intensive workload. The workload generator generates a request rate that follows a Poisson distribution to simulate a number of concurrent users.

We ran our experiments in private VMware ESXi [17] testbed. Figure 2 shows software stack, hardware specification, and sample topology of our experiments. A three-digit notation $\#Web/\#App/\#DB$ is used to denote the number of web servers, app servers, and DB servers deployed in the experiments. For example, Figure 2(b) shows a 1/1/1 sample topology, which includes one Apache, one Tomcat, and one MySQL. Each server in the system is deployed in a VM which is hosted in a dedicated physical node. For each hardware sample topology, we evaluate three representative soft resources–the thread pool in a web server, the thread pool and the DB connection pool [2] in an app server. These three soft resources restrict the maximum number of concurrent processing requests in Apache, Tomcat, and

[2]The size of DB connection pool in an app server will yield the maximum concurrent requests flowing to the downstream DB tier.


(a) Tomcat (1-core) achieves the highest throughput when workload concurrency is 10.


(b) Tomcat (2-core) achieves the highest throughput when concurrency is 20.

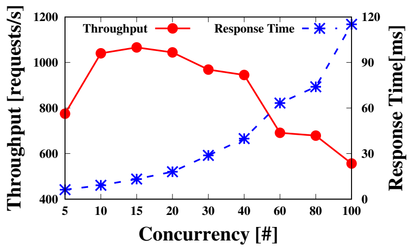

(c) Tomcat (2-core) achieves the highest throughput when concurrency is 15 after we double the dataset size in database.

Fig. 3: Performance variation at increasing workload concurrency for Tomcat in a 3-tier system. Figure 3(a) and 3(b) show that changing the # of CPU cores could change the optimal concurrency setting in Tomcat. Figure 3(b) and 3(c) shows the system state variation (e.g., dataset size change in MySQL) could also affect the optimal concurrency setting in Tomcat.

MySQL, respectively. We denote such three soft resources as $\#W_{threads}$-$\#A_{threads}$-$\#DB_{connections}$. Both hardware resource utilization and application-level metrics (e.g., throughput) measurements are taken during the runtime period at a fine granularity (e.g., 50ms).

B. Quantitative Analysis for Optimal Concurrency Setting with Pre-profiling Condition Variation

In this section, we show a quantitative analysis of optimal concurrency settings for typical servers in a 3-tier system with various pre-profiling conditions. Previous research [10], [12] demonstrates that the request processing concurrency would affect the efficiency of servers, which significantly impacts the server performance. Consequently, inappropriate concurrency settings for the specific server would incur performance degradation such as large response time fluctuations and significant throughput drop. To generate the optimal concurrency setting for component servers, offline profiling on various concurrency workloads through a queueing network model is widely adopted by academic research [10] and industry practitioners. However, our experimental results in the following sections show that many factors (e.g., the system state, workload characteristic, and scaling strategy) would cause the pre-profiling condition varies significantly, leading to sub-optimal performance after the system scaling.

We conduct extensive experiments to explore the impact of various pre-profiling conditions on the optimal concurrency settings of component servers in web applications, as shown in Figure 3. In this set of experiments, we use a modified RUBBoS workload generator to send the browse-only requests with zero think time between consecutive requests, thus we can precisely control the request processing concurrency by specifying the number of threads to stress the target server (e.g., Tomcat or MySQL). For each controlled concurrency level, we configure the same concurrency setting for the corresponding server to avoid queue overflow. Concretely, we adjust the thread pool size and database connection pool size in Tomcat to control the concurrency level in Tomcat and MySQL, respectively.

We show two types of pre-profiling condition variations which are common in the production environment. First, we examine the optimal concurrency variation when scaling strategy changes from scale-out to scale-up (add or remove # of CPU cores). Figure 3(a) and 3(b) show that Tomcat with 1-core achieves the optimal performance when concurrency is 10, however optimal concurrency setting for Tomcat in a 2-core scenario changes to 20. Such experimental results indicate that vertical scaling would cause the original concurrency optimal setting to be sub-optimal.

The second pre-profiling condition is the change of system state. We use different dataset size to describe such variation. In the production environment, as the dataset size varies along with continuous dataset updating, it would affect the sensitivity of the application server (e.g., Tomcat) to concurrency level. Comparing Figure 3(b) with 3(c), even for server with the same critical computing resources(e.g., # of CPU cores) experiencing the same type of workload, the optimal concurrency changes from 20 to 15 after we manually enlarging the dataset size. Consequently, the optimal concurrency range for MySQL would also change when the types of workload vary.

Our previous results demonstrate that the changes of pre-profiling conditions have a significant impact on the soft resource allocation of component servers in web applications; only scaling hardware resources using the soft resource allocation based on the static pre-profiling results without considering the changes of pre-profiling conditions could lead to inferior performance. Considering the naturally bursty workload and unpredictable system state of web applications, fast runtime adapting soft resources is required on the web application scaling management.

III. Scatter-Concurrency-Throughput Model

In this section, we propose an online Scatter-Concurrency-Throughput (SCT) Model which fast determines an optimal soft resource allocation for the component server of an n-tier application. Our model extends the classic statistical intervention analysis for the bottleneck detection [18] with a significant change: we take the non-trivial multithreading overhead of each component server in the system into consideration, which is non-negligible when facing high request processing concurrency. The goal of our model is to stabilize response time fluctuations and achieve high throughput by estimating the optimal concurrency setting for component servers on the fly.

A. Model Description

Based on the classic Utilization Law [13], a server's throughput increases as the workload concurrency increases until the server reaches saturation. By then further increasing the workload concurrency only leads to throughput plateau or even throughput decrease due to non-trivial multithreading overhead as shown by plenty of previous research studies [10], [19]–[21]. To better illustrate our SCT model, Figure 4 characterizes the relationship between a server's throughput and its concurrency using the following three server stages.

Ascending Stage. When the server concurrency is relatively low, the throughput increases almost linearly as the concurrency increases until it reaches the maximum throughput TP_{max}. Theoretically, a server reaches the maximum throughput when its bottleneck resource is 100% utilized; the value of the maximum throughput is determined by the average demand for the bottleneck resource per job. We define the minimum concurrency of a server achieving the maximum throughput as the `lower bound` of a rational soft resource allocation for the server, which we denote as Q_{lower}.

Stable Stage. In this stage, the server maintains the maximum throughput (i.e., TP_{max}) when the workload concurrency continues to increase beyond Q_{lower}. We define the maximum concurrency that the server maintains its maximum throughput as the `upper bound` of a rational soft resource allocation for the server, which we denote as Q_{upper}.

Descending Stage. When the server concurrency exceeds the Q_{upper}, the server throughput enters the descending stage. This is because component servers like MySQL of an n-tier system widely adopt a thread-based synchronous RPC-style mechanism for inter-tier communications. It means that processing one client request requires one dedicated thread in each server in the system [12]. Previous research studies [10], [19]–[21] show that thread-based servers under high workload concurrency suffer from non-trivial multithreading overhead caused by various factors such as increased lock contention among contending threads, crosstalk penalty due to cache coherence requirement [21], and Java garbage collection due to increased memory usage and activities [12]. Such multithreading overhead incurs non-linear throughput degradation when the server concurrency increases to a high level.

Our Scatter-Concurrency-Throughput (SCT) model estimates the rational concurrency range (i.e., $[Q_{lower}, Q_{upper}]$) based on a statistical analysis of each server's real-time throughput, response time, and concurrency. Figure 4 shows an overview of the SCT model and illustrates the workflow of online optimal concurrency setting estimation. There are two major phases in SCT: the Real-time Metrics Collection phase and the rational concurrency range Estimation phase. In the Real-time Metrics Collection phase, we collect a series of tuples $\{Q_{tn}, TP_{tn}, RT_{tn}\}$ during a short time window (e.g., 3 minutes). Each tuple consists of a server's real-time concurrency, throughput, and response time. Since a server's

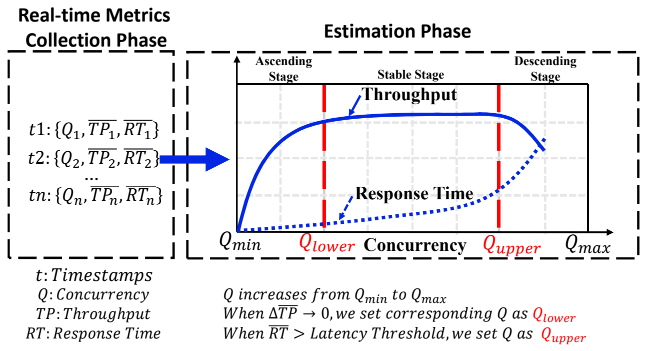
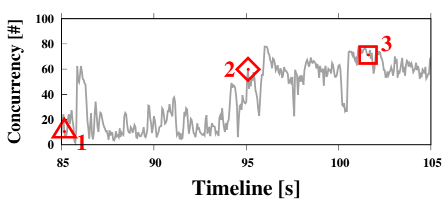

Real-time Metrics Collection Phase
Estimation Phase
$t1: \{Q_1, \overline{TP_1}, \overline{RT_1}\}$
$t2: \{Q_2, \overline{TP_2}, \overline{RT_2}\}$
...
$tn: \{Q_n, \overline{TP_n}, \overline{RT_n}\}$
Ascending Stage
Stable Stage
Descending Stage
Throughput
Response Time
Q_{min}
Q_{lower}
Concurrency
Q_{upper}
Q_{max}
t: Timestamps
Q: Concurrency
TP: Throughput
RT: Response Time
Q increases from Q_{min} to Q_{max}
When $\Delta\overline{TP} \to 0$, we set corresponding Q as Q_{lower}
When $\overline{RT}$ > Latency Threshold, we set Q as Q_{upper}

Fig. 4: Overview of SCT Model and its workflow for rational concurrency range estimation.

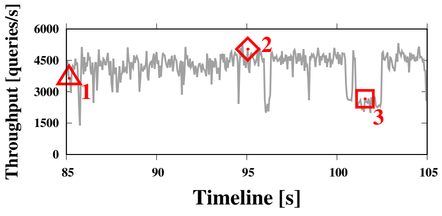
real-time concurrency under real-world workload varies significantly, we denote the server concurrency variation during each time window as $[Q_{min}, Q_{max}]$. For each specific server concurrency $Q_n (Q_n \in [Q_{min}, Q_{max}])$, we calculate the average throughput $\overline{TP_n}$ and average response time $\overline{RT_n}$. Finally, it would provide a series of data tuple $\{Q_n, \overline{TP_n}, \overline{RT_n}\}$ for the estimation phase. In the estimation phase, we apply statistic intervention analysis [18] to process sufficient data tuples, and we can successfully estimate the rational concurrency range $[Q_{lower}, Q_{upper}]$ of a server as shown in Figure 4. We note that during this rational concurrency range (i.e., Stable Stage), the server is able to maintain its highest throughput. However, considering strict SLA requirements (e.g., bounded response time) for web applications, we choose the lower bound Q_{lower} as the optimal concurrency setting for each component server since lower concurrency means lower response time (see the dash line in Figure 4).

B. Fine-grained Throughput and Concurrency Analysis

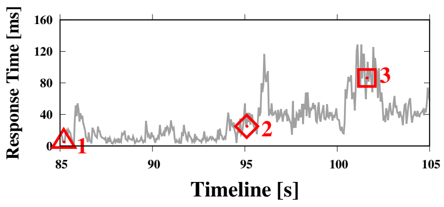
To correctly characterize the performance of a server in an n-tier web application under various concurrency workloads, one of the key points is the continuous fine-grained measurement of a server's throughput and concurrency in a certain time interval. Concretely, we calculate the throughput by counting how many requests are completed in the server within a fixed time interval (e.g., 50ms); the response time and the concurrency of a server are calculated by the average response time of completed requests and the number of concurrent processing requests within the same time interval, respectively. It is important to choose an appropriate time interval to measure the throughput and concurrency of a server in the system in the runtime. Too long or too short of the time interval would bring side-effects on estimating the optimal concurrency range. We set the time interval to be 50ms, which is a reasonable setting in our experiments. Figure 5(a), 5(b), and 5(c) show the MySQL concurrency, throughput, and response time measured at the same 50ms time interval over a 20-second runtime after the system scales from 1/1/1 to 1/2/1 due to increased workload (i.e., period 85s$\sim$105s in Figure 1). We note that MySQL is the single bottleneck in the system after the system scaling. Our experimental results show that the



(a) MySQL workload concurrency.



(b) MySQL throughput.



(c) MySQL response time.

Fig. 5: Fine-grained monitoring of MySQL when the 3-tier system serves a realistic bursty workload. Figure 5(a), 5(b), and 5(c) show MySQL's "real-time" (measured at 50ms time granularity) concurrency, throughput, and response time within the same 20-second experiment period, respectively. Three points are chosen to illustrate their position in the scatter graphs (the correlation analysis between metrics) in Figure 6.

MySQL concurrency, throughput, and response time fluctuate significantly after a new Tomcat is added into the system due to the hardware-only scaling, which indicates less CPU efficiency in MySQL.

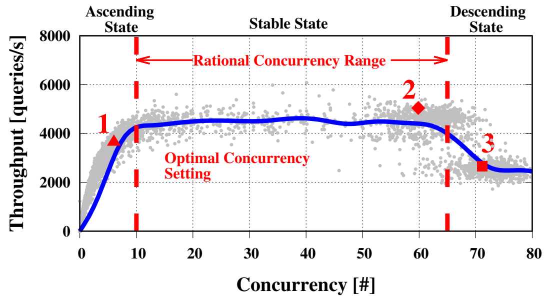
We further correlate the MySQL's fine-grained concurrency, throughput, and response time for quantitative analysis of the impact of the request processing concurrency on the server performance (e.g., throughput and response time) as shown in Figure 6 [3]. Figure 6(a) shows the correlation between the MySQL concurrency and throughput, which is derived from Figure 5(a) and 5(b). Each point in this figure represents the MySQL concurrency and throughput measured at the same 50ms time interval during a 20-second runtime. We also correlate the MySQL concurrency and response time measured at the same 50ms time interval as shown in Figure 6(b). The phenomenon that the MySQL throughput in Figure 6(a) has distict three states as the concurrency increasing validates our previous analysis in Section III-A.

Through our fine-grained analysis on the performance of a server in the system under a realistic workload, we note that too small (e.g., $< Q_{lower}$) or too large (e.g., $> Q_{upper}$) of the request processing concurrency will lead to large performance fluctuations of a server in the system, which is due to less server CPU efficiency. Consequently, to achieve both good performance and high resource efficiency during the system scaling phase, we should consider a proper adaption of soft resources in the system, which controls the server request processing concurrency.

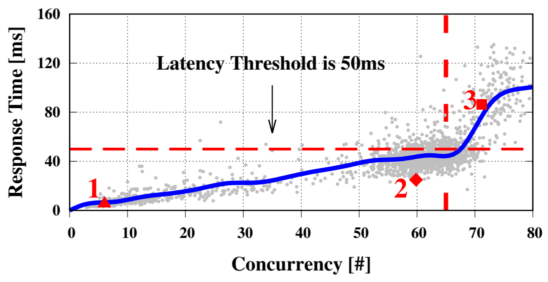
C. Factors that Affect Optimal Concurrency Setting

In this section, we further present an empirical study on three runtime environment changes that affect the optimal concurrency setting of different servers in web application: different hardware scaling strategies, the change of system state, and the change of workload characteristics. Auto-Scaling solutions with static concurrency settings are incapable of handling these changes due to rapid variations of the optimal concurrency setting during the runtime. However, we will demonstrate how our proposed SCT model deals with these three changes during the system scaling phase.

[3]To display the smoothed curve, we use the `smooth` function in Gnuplot [22] for data smoothing with the cubic-splines or the Bezier curves.



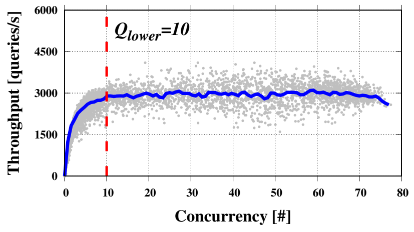
(a) The scatter graph shows the correlation between "real-time" server throughput and concurrency.



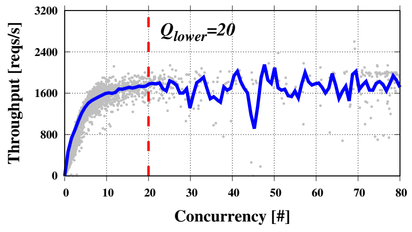
(b) The scatter graph shows the correlation between "real-time" server response time and concurrency.

Fig. 6: The correlations between MySQL concurrency, throughput, and response time measured at 50ms granularity during a 12-minute experiment. The blue line is the trend line for each scatter graph. These two scatter graphs help determine a rational range of concurrency setting for MySQL. We choose the lower bound of the rational range as the optimal concurrency setting (see Figure 6(a)) since it achieves the highest server throughput and the minimum response time within the range.

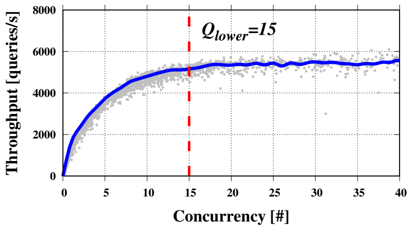
1) Scaling Strategy: Horizontal vs Vertical. The vertical scaling, i.e., adding or removing resources (e.g., CPU and memory) would affect the optimal concurrency setting of a component server in n-tier applications. We start our experiments with 1/4/1 hardware topology and browse-only CPU-intensive workload. MySQL is the single bottleneck server in the system. We initially configure MySQL with 1-core CPU and collect concurrency and throughput metrics in a 50ms granularity, then we scale up MySQL CPU to 2-core. Fig-



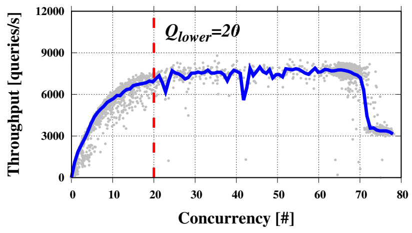
(a) The 1-core MySQL server case.



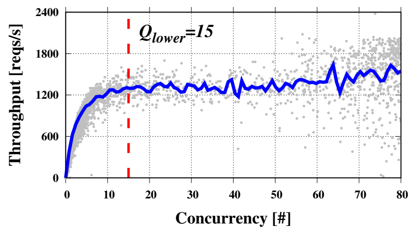
(b) The Tomcat server case with the original RUBBoS dataset.



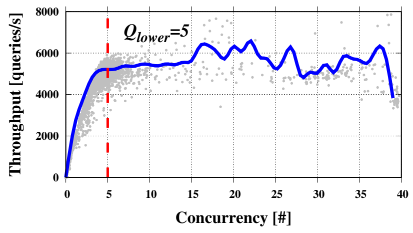
(c) The MySQL server case under CPU-intensive workload.



(d) The 2-core MySQL server case.



(e) The Tomcat server case with the enlarged RUBBoS dataset case.



(f) The MySQL server case under read-write-mix I/O intensive workload.

Fig. 7: The comparison between server throughput-concurrency scatter graphs after vertical scaling, RUBBoS dataset size change, and workload characteristics change. Figure 7(a) and 7(d) show the impact of vertical scaling (e.g., 1-core to 2-core). Figure 7(b) and 7(e) show the impact of system state change (e.g., dataset size). Figure 7(c) and 7(f) show the impact of workload characteristics change. These case studies show that the scatter graph is able to precisely capture the shift of the optimal server concurrency setting once the system environment changes.

ure 7(a) and 7(d) show the concurrency-throughput correlation comparison of MySQL after the MySQL server scales up from 1-core to 2-core. For example, with the help of the SCT model, we observe that the lower bound of the rational concurrency setting Q_{lower} doubles from 10 to 20 after the MySQL scaling up, indicating the optimal concurrency setting variation of the MySQL server due to vertical scaling. An interesting phenomenon is that horizontal scaling would not cause such variation (details are omitted due to space constraints).

2) System State: Original RUBBoS Dataset vs Manually Enlarged Dataset. The system state variation affects the optimal concurrency setting of a component server in an n-tier application by affecting the degrees of computation for the business logic for the same type of client requests. For example, in the production environment, the permanent datasets of web applications vary all the time due to continuous dataset updates, which leads to variations of the service time of client requests accordingly. In this set of experiments, we start our experiments with 1/1/4 hardware topology and browse-only workload. Tomcat is the single bottleneck server in the system. We initially employ the original RUBBoS dataset, and then we manually enlarged the original RUBBoS dataset. Figure 7(b) and 7(e) show the concurrency-throughput correlation comparison of Tomcat after we enlarge the dataset size in MySQL. Concretely, our experimental results show that the lower bound of the optimal soft resource Q_{lower} decreases from 20 to 15 after the change of dataset size, which helps explain the extensive experimental results in Section II-B.

3) Workload Type: CPU-intensive vs I/O intensive. Furthermore, we explore the impact of different workload types on the optimal soft resource allocation of a component server in n-tier applications. We switch the workload from read-only CPU-intensive mode to read/write-mix I/O-intensive mode. We start our experiments with 1/4/1 hardware topology. MySQL is the single bottleneck in the system, and the critical hardware resources change from CPU to disk I/O due to the change of the browse-only workload to the read/write-mix I/O-intensive workload, leading to significant variations of the capacity of the server. Figure 7(c) and 7(f) show the concurrency-throughput correlation comparison of MySQL between the CPU-intensive workload and the I/O-intensive workload. We observe that the lower bound of the optimal soft resource Q_{lower} decreases from 15 to 5 after we change the read-only "ViewStory" workload to the read/write-mix I/O-intensive "StoreStory" workload.

Previous experimental results demonstrate that our proposed SCT model is capable of capturing the optimal concurrency setting variation of a component server when the system encounters the runtime environment changes (e.g., the hardware scaling strategies, the system state, and the workload characteristics).

IV. Concurrency-aware System Scaling Design and Implementation

So far we have discussed our SCT model to fast estimate the optimal concurrency settings of a component server in an n-tier application during runtime. Our previous experimental results show that the hardware-only scaling solutions are not able to handle the optimal concurrency setting variations due to the runtime environment changes during system scaling phase (see Figure 1); thus an appropriate adaption of soft resources

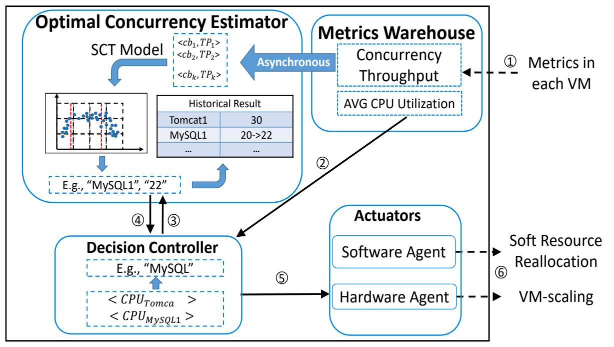


Fig. 8: The architecture of our ConScale framework.

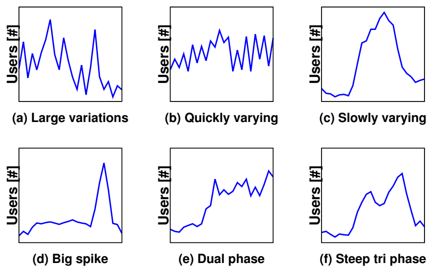


Fig. 9: Realistic workload traces used in our experiments.

is important regarding the dynamic runtime environment to achieve good performance while maintaining high resource efficiency. In this section, we propose our Concurrency-aware system Scaling (ConScale) framework, which integrates the SCT model (see Section III) and can fast conduct an optimal soft resource adaption in the bottleneck tier of the system on the fly. Figure 8 shows that our ConScale framework consists of four components: Metric Warehouse, Online Optimal Concurrency Estimator, Decision Controller, and Actuators.

Metric Warehouse collects both application- and system-level metrics (e.g., throughput and CPU) through the monitoring agents installed in each VM at every one second (step 1 in Figure 8) and provides the data to Decision Controller and Optimal Concurrency Estimator. **Decision Controller** decides when and how to turn on/off VMs based on the system need and retrieves the optimal concurrency setting from Optimal Concurrency Estimator. **Actuators** arrange VM-scaling and soft resource reallocation according to the demand of the Decision Controller. **Optimal Concurrency Estimator** has another workflow, it continuously pulls fine-grained application-level metrics (e.g., concurrency and throughput) from the metric warehouse asynchronously, and feeds the data to the SCT model as we introduced in Section III to generate an optimal concurrency setting of a server in the system.

A. *Implementations*

VM-scaling. Since the underlying hypervisor in clouds already provides sufficient APIs to manipulate VMs, we can easily launch and turn off VMs via calling these APIs remotely. However, there are two main challenges for VM scaling. First, it is a non-trivial task to add new VMs running stateful servers (e.g., DB server) due to the complicated data/state consistency problem [23]. To solve this problem, we simply replicate the MySQL dataset in the scaling phase. Due to the small dataset size of RUBBoS, we set a 15-second preparation period before launching each new VM, which is considered as a reasonable configuration in our experiments. Longer preparation period could be required due to more complex data/state consistency in a real production environment.

Second, we need to consider the load balancing problem among the original servers in the system with the newly added servers after scaling. In our experiments, we use HAProxy [15], which provides both HTTP and TCP-based proxying, as a load balancer for both the application tier and the DB tier. Concretely, a new incoming request from either the web tier or the application tier will be dispatched by HAProxy to the downstream tier based on a pre-configured load balancing policy (e.g., `roundrobin` or `leastconn`). In our implementation, we adopt `leastconn` policy.

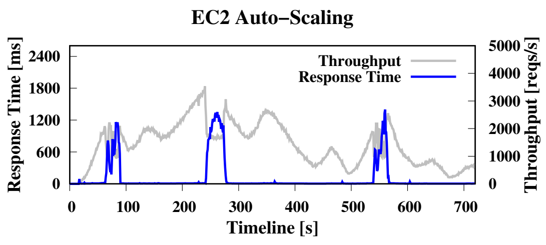
Soft resource adaption. After hardware scaling, the Decision Controller automatically adapts soft resources in the system to yield the request processing concurrency of each related server. In the current implementation, we dynamically adjust the thread pool size to limit the request processing concurrency of the Tomcat server; and we control the DB connection pool size in the Tomcat server to restrict the maximum request processing concurrency of MySQL.

Concretely, the runtime adaption of the thread pool size in Tomcat is implemented in the latest Tomcat [24] through Java Management Extensions (JMX) [25] technology. In this case, our software agent is able to fetch or modify the thread pool size via RMI (Remote Method Invocation). However, the Tomcat server JMX service does not provide the ability to change the size of the database connection pool in the runtime. Therefore, we extend the JMX service in Tomcat to expose the interfaces of the DB connection pool size management.

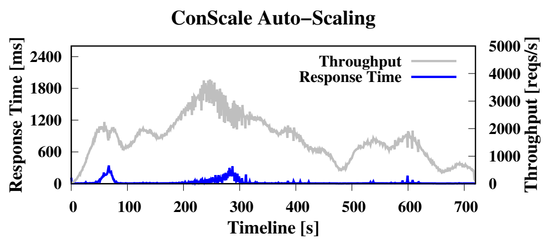
V. EXPERIMENT EVALUATION

We evaluate the effectiveness of our ConScale in stabilizing performance fluctuations under six realistic bursty workloads compared to two state-of-the-art scaling mechanisms: Amazon AWS service EC2-AutoScaling [9] and the concurrency-aware DCM [10] framework. We will show that ConScale outperforms the other two scaling mechanisms in achieving high throughput and stabilizing the system response time because of fast soft resources adaption during the temporary overloading.

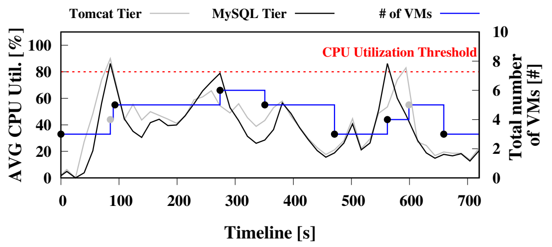
Experimental Setup. We implement and deploy three scaling frameworks (i.e., EC2-AutoScaling, DCM, and ConScale) in our private VMware ESXi cluster. The hardware-only scaling framework EC2-AutoScaling is widely-adopted in academic research and industry practices, which maintain customers' applications by automatically adding or removing VMs according to the pre-defined conditions (e.g., the server CPU utilization $> 80\%$). The concurrency-aware DCM framework integrates the concurrency-aware model to intelligently reallocate soft resources in the system during the system scaling process. To avoid performance instability problem, we



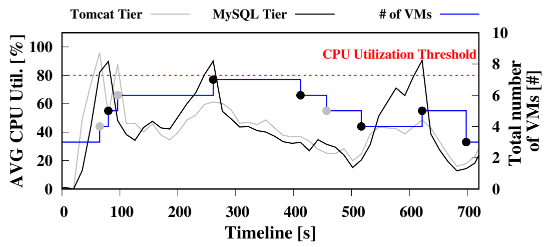
(a) The system response time encounters large fluctuations under the "Large Variation" workload and the system throughput drops during the response time spikes at 62s, 244s and 545s.



(b) The system response time is stable and low under the same workload in (a) and the system throughput matches the workload variations as shown in Figure 9(a).



(c) The Tomcat/MySQL tier scales out/in along with the CPU utilization. Tomcat scales out at timeline 85s and 595s while MySQL scales out at timeline 90s, 273s, and 552s.



(d) The Tomcat/MySQL tier scales out/in along with the CPU utilization. Tomcat scales out at timeline 65s and 96s, while MySQL scales out at timeline 85s, 261s, and 622s.

Fig. 10: Large performance fluctuations of EC2-AutoScaling compared to ConScale using the same "Large Variation" workload trace. The left-side figures show the performance of EC2-AutoScaling while the right-side figures show the performance of ConScale. Both systems start with the same system topology 1/1/1 and the soft resource allocation 1000-60-40. However, ConScale outperforms EC2-AutoScaling once the system scales.

TABLE I: Tail response time (i.e., 95th and 99th percentile response time) comparison between EC2-AutoScaling and ConScale under six realistic bursty workload traces.

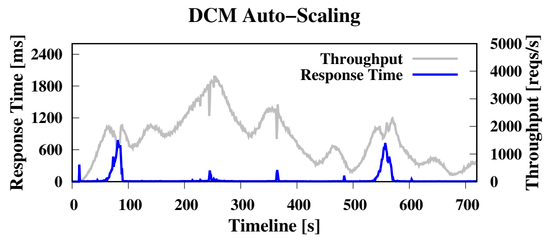
Percentile Response Time [ms]		*Large Variation*	*Quick Varying*	*Slowly Varying*	*Big Spike*	*Dual Phase*	*Steep Tri Phase*
RT_{95th}	*EC2-AutoScaling*	462	157	1135	687	225	101
	ConScale	**157**	**48**	**85**	**179**	**81**	**56**
RT_{99th}	*EC2-AutoScaling*	2345	684	3252	3981	1153	1259
	ConScale	**465**	**229**	**218**	**479**	**328**	**171**

adopt the "quick start but slow turn off" hardware scaling strategy to avoid performance instability problem [6]. We conduct our evaluation experiments of three frameworks under six realistic bursty workload traces, as shown in Figure 9. The workload traces are collected from real-world traces and further categorized by Gandhi [6]. We ran our experiments with the 7500 maximum concurrent users for 12 minutes.

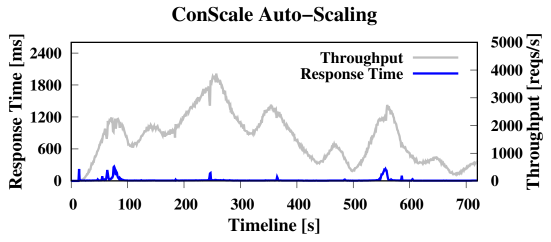
Performance Comparison between ConScale and EC2-AutoScaling. Figure 10 shows the performance comparison between ConScale and EC2-AutoScaling frameworks under the same "Large Variation" workload (See Figure 9). The left two figures (Figure 10(a) and 10(c)) show the EC2-AutoScaling case, and the right two figures (Figure 10(b) and 10(d)) show our ConScale case. In this set of experiments, we start our evaluation with 1/1/1 hardware topology and soft resource allocation is 1000-60-40.

The system with both ConScale and EC2-AutoScaling scales when the CPU usage of either the Tomcat tier or the MySQL tier exceeds a pre-defined threshold (i.e., 80%). Comparing Figure 10(a) and 10(b), our ConScale has relatively stable response time and throughput in a 12-minute experiment runtime than that in the EC2-AutoScaling case. For example, EC2-AutoScaling encounters large response time fluctuations and significant throughput drop at the scaling out phase (periods 62s∼95s, 244s∼285s, and 545s∼570s). Taking the first period 62s∼95s in Figure 10(a) for example, EC2-AutoScaling shows a significant throughput degradation and a large response time spike. We observe that a new Tomcat is added into the system at 85s because the CPU utilization of the Tomcat tier exceeds a pre-defined threshold (Figure 10(c)). Once the second Tomcat starts to serve incoming requests, MySQL becomes the new bottleneck tier and the Tomcat tier now is able to send double concurrent requests to the downstream MySQL. High concurrent requests in MySQL cause low efficiency of MySQL CPU (see Figure 6). On the other hand, there is moderate performance degradation during the system scaling process in our ConScale case in Figure 10(b). This is because our ConScale framework estimates the optimal concurrency setting of both Tomcat and MySQL based on the SCT model during the temporary overloading, and such resource allocation could guarantee the server can fully and efficiently utilize the hardware resources and achieves much more stable performance compared to the EC2-AutoScaling case.

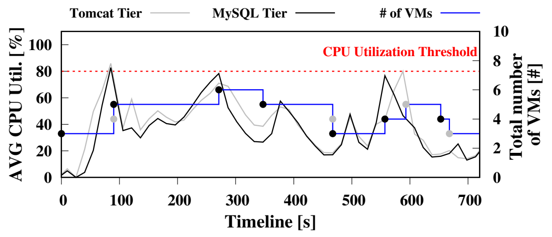
We further compare tail response time (i.e., 95th and 99th percentile) between EC2-AutoScaling and ConScale frameworks under other types of workload traces (see Figure 9) in Table I. Our results show that even for the 99th percentile



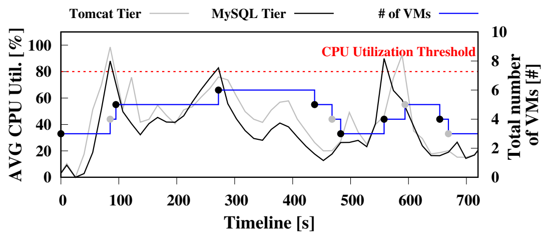
(a) The system response time encounters large fluctuations under the “Large Variation” workload at timeline 56s and 556s.



(b) The system response time is stable and low under the same workload in (a).



(c) The Tomcat/MySQL tier scales out/in along with the CPU utilization. Tomcat scales out at timeline 85s and 590s while MySQL scales out at timeline 85s, 272s, 565s.



(d) The Tomcat/MySQL tier scales out/in along with the CPU utilization. Tomcat scales out at timeline 75s and 590s while MySQL scales out at timeline 98s, 273s, and 566s.

Fig. 11: Our proposed ConScale framework achieves much more stable and low response time and higher throughput than that in the DCM case when the system state changes (i.e., the dataset size). We reduce the dataset size in database to simulate system state change in the production environment.

response time, ConScale can still limit the response time below 500ms for all the cases, which is required for most web applications [26]

Performance Comparison between ConScale and DCM. Here we compare the performance between ConScale and DCM to demonstrate the adaptiveness and reliability of our online Scatter Concurrency-Throughput (SCT) model compared to the offline queuing network model with Concurrency-Aware model which requires a training process for specific workload [10]. One intuitive drawback for the offline models is that they require additional offline training for the optimal soft resource allocation when the runtime environment (e.g., scaling strategy, system state, and workload type) changes as we discussed in Section III-C.

We start our experiment using the “Large Variation” workload with browse-only CPU-intensive mode. At first, we use the original RUBBoS dataset to train DCM for the optimal concurrency setting, which gives us 20 for Tomcat thread pool size and 40 for MySQL DB pool size. We then compare the performance of two frameworks (DCM and ConScale) using the same workload with a manually reduced dataset to simulate the system state change due to continuous dataset updates in a real-world production system. According to the analysis in Section III-C, the optimal concurrency setting should vary. Based on our model, ConScale estimates that the new optimal concurrency setting for Tomcat should be 30. Figure 11 shows that our proposed ConScale framework achieves much more stable low response time and higher throughput than that in DCM case during the temporary overloading since it adjusts concurrency setting for Tomcat. Taking the first response time spike (85s$\sim$90s) in Figure 11(a) for example, it is because that DCM is unaware of the change of the system state, the original “optimal” setting becomes too low and causes the server cannot fully utilize the hardware resources (e.g., CPU) (the under-allocation effect [12]). Furthermore, it leads to the low efficiency of the Tomcat CPU and sub-optimal system performance. Consequently, such a phenomenon validates the accuracy of the estimation of our online SCT model.

VI. Related Work

Threshold-based Auto-Scaling Technique is widely used among industry cloud providers for computing resource management. One of the key points of threshold-based auto-scaling is how to decide the threshold to achieve good performance. For example, Dutreilh et al. [27] avoid scaling actions oscillations in the system by controlling the cool-down period and the inertia interval. Hasan et al. [28] correlate multiple metrics (e.g., CPU, storage, and network) to construct a more detailed threshold. Similarly, our work utilizes hybrid metrics consists of CPU Utilization, server concurrency and, throughput to guarantee the robustness of threshold.

Auto-Scaling Technique with Adaptive Resource Allocation. Previous work [29], [30] shows that the complex dependencies among different tiers are the main performance problem of an n-tier system under the bursty workload. Many research efforts advocate that the adaptive resource allocation strategy is an effective solution to improve the scalability in clouds. Nathuji et al. [29] consider performance interference effects due to the application consolidation in clouds and develop a QoS-aware control framework, Q-clouds, to transparently adjust resource allocations to guarantee the QoS requirement. Sun et al. [30] propose a model-based framework, ROAR, to optimize and automate cloud resource allocation decisions to meet the QoS goal. Our work focuses on improving the adaptivity of the auto-scalers by fast and intelligent soft resources adaption to match the hardware resource variation in system scaling scenarios in clouds.

VII. Conclusion

We studied the importance of the online estimation of the optimal concurrency setting for component servers in an n-tier system. It is used to realize dynamic concurrency reallocation to mitigate large response time fluctuations for the system during the scaling phase in clouds. Through our standard n-tier benchmark RUBBoS experiments under the realistic workload, we reveal that several factors that may incur rational concurrency variation (Section II-B and Section III-C). We then propose a Scatter-Concurrency-Throughput (SCT) model using fine-grained measurement data to estimate a rational concurrency setting of a server in the system on the fly (Section III). We further develop a Concurrency-aware system Scaling (ConScale) framework which integrates the aforementioned SCT model, with a goal of fast and dynamically adapting soft resources of each tier during the system scaling phase (Section IV). Our evaluation experiments under six representative realistic workloads show that ConScale can effectively stabilize large performance fluctuations and reduce the long-tail latency compared to the state-of-the-art hardware-only auto-scaling technique and dynamic concurrency reconfiguration framework [10] (Section V).

VIII. ACKNOWLEDGMENT

This research has been funded by National Science Foundation by CISE's CNS-1566443 and CNS-2000681. 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 mentioned above.

References

[1] P. Bodik, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson, "Characterizing, modeling, and generating workload spikes for stateful services," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 241–252.

[2] B. Snyder, "Server virtualization has stalled, despite the hype," *InfoWorld*, 2010.

[3] A. Sriraman and T. F. Wenisch, "μtune: Auto-tuned threading for {OLDI} microservices," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 177–194.

[4] B. Vamanan, J. Hasan, and T. Vijaykumar, "Deadline-aware datacenter tcp (d2tcp)," *ACM SIGCOMM Computer Communication Review*, 2012.

[5] R. Rojas-Cessa, Y. Kaymak, and Z. Dong, "Schemes for fast transmission of flows in data center networks," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 3, pp. 1391–1422, 2015.

[6] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch, "Autoscale: Dynamic, robust capacity management for multi-tier data centers," *ACM Transactions on Computer Systems (TOCS)*, 2012.

[7] R. Han, L. Guo, M. M. Ghanem, and Y. Guo, "Lightweight resource scaling for cloud applications," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. IEEE Computer Society, 2012, pp. 644–651.

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

[9] A. W. Services, "Amazon ec2 auto scaling," https://aws.amazon.com/ec2/autoscaling/, 2019.

[10] Q. Wang, H. Chen, S. Zhang, L. Hu, and B. Palanisamy, "Integrating concurrency control in n-tier application scaling management in the cloud," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 4, pp. 855–869, 2018.

[11] T. Lorido-Botrán, J. Miguel-Alonso, and J. A. Lozano, "Auto-scaling techniques for elastic applications in cloud environments," *Department of Computer Architecture and Technology, University of Basque Country, Tech. Rep. EHU-KAT-IK-09*, vol. 12, p. 2012, 2012.

[12] Q. Wang, S. Malkowski, Y. Kanemasa, D. Jayasinghe, P. Xiong, C. Pu, M. Kawaba, and L. Harada, "The impact of soft resource allocation on n-tier application scalability," in *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2011, pp. 1034–1045.

[13] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., 1984.

[14] O. Consortium, "Rubbos: Bulletin board benchmark," http://jmob.ow2.org/rubbos.html, 2005.

[15] H. Community, "Haproxy: The reliable, high performance tcp/http load balancer," http://www.haproxy.org/, 2019.

[16] Dormando, "Memcached," https://memcached.org/, 2019.

[17] VMware, "Vmware esxi: The purpose-built bare metal hypervisor," https://www.vmware.com/products/esxi-and-esx.html, 2019.

[18] S. Malkowski, M. Hedwig, J. Parekh, C. Pu, and A. Sahai, "Bottleneck detection using statistical intervention analysis," in *International Workshop on Distributed Systems: Operations and Management*. Springer, 2007, pp. 122–134.

[19] M. Welsh, D. Culler, and E. Brewer, "Seda: an architecture for well-conditioned, scalable internet services," in *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5. ACM, 2001, pp. 230–243.

[20] Y. Diao, J. L. Hellerstein, S. Parekh, H. Shaikh, and M. Surendra, "Controlling quality of service in multi-tier web applications," in *26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)*. IEEE, 2006, pp. 25–25.

[21] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[22] W. Thomas and K. Colin, "gnuplot homepage," http://www.gnuplot.info/, 2019.

[23] 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.

[24] A. S. Foundation, "Apache tomcat 7," https://tomcat.apache.org/tomcat-7.0-doc/funcspecs/mbean-names.html, 2019.

[25] Oracle, "Java management extensions (jmx) technology," https://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html, 2019.

[26] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[27] X. Dutreilh, A. Moreau, J. Malenfant, N. Rivierre, and I. Truck, "From data center resource allocation to control theory and back," in *2010 IEEE 3rd international conference on cloud computing*. IEEE, 2010.

[28] M. Z. Hasan, E. Magana, A. Clemm, L. Tucker, and S. L. D. Gudreddi, "Integrated and autonomic cloud resource scaling," in *2012 IEEE network operations and management symposium*. IEEE, 2012, pp. 1327–1334.

[29] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds: managing performance interference effects for qos-aware clouds," in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 237–250.

[30] Y. Sun, J. White, S. Eade, and D. C. Schmidt, "Roar: A qos-oriented modeling framework for automated cloud resource allocation and optimization," *Journal of Systems and Software*, vol. 116, pp. 146–161, 2016.