Integrating Concurrency Control in n-Tier Application Scaling Management in the Cloud

Qingyang Wang, *Member, IEEE,* Hui Chen, *Member, IEEE,* Shungeng Zhang, *Member, IEEE,* Liting Hu, *Member, IEEE,* Balaji Palanisamy, *Member, IEEE,*

Abstract—Scaling complex distributed systems such as e-commerce is an importance practice to simultaneously achieve high performance and high resource efficiency in the cloud. Most previous research focuses on hardware resource scaling to handle runtime workload variation. Through extensive experiments using a representative n-tier web application benchmark (RUBBoS), we demonstrate that scaling an n-tier system by adding or removing VMs without appropriately re-allocating soft resources (e.g., server threads and connections) may lead to significant performance degradation resulting from implicit change of request processing concurrency in the system, causing either over- or under-utilization of the critical hardware resource in the system. We build a concurrency-aware model that determines a near optimal soft resource allocation of each tier by combining some operational queuing laws and the fine-grained online measurement data of the system. We then develop a dynamic concurrency management (DCM) framework that integrates the concurrency-aware model to intelligently reallocate soft resources in the system during the system scaling process. We compare DCM with Amazon EC2-AutoScale, the state-of-the-art hardware only scaling management solution using six real-world bursty workload traces. The experimental results show that DCM achieves significantly shorter tail latency and higher throughput compared to Amazon EC2-AutoScale under all the workload traces.

Index Terms—scalability, soft resources, configuration, web application, parallel processing, cloud computing

1 Introduction

An important feature of cloud computing platforms is scalability, the ability to scale system resources for both high performance and high resource efficiency. Such ability is especially important for web applications such as e-commerce because of two reasons. First, web applications in general adopt the n-tier architecture (e.g., web tier, application server tier, and the database tier; other tiers such as load balancer and Memcached [1] are also common), the capacity of each tier is supposed to be easily scaled by adding or removing server VMs. Second, workload for web applications is naturally bursty. For example, the number of users accessing an e-commerce website (e.g., Amazon.com) can be over 10X larger in rush hours (e.g., black Friday) than that in normal periods. The traditional strategy of static provisioning always for peak workload will lead to significant waste of computing resources and power consumption. So to achieve both high performance and high resource efficiency, it is extremely important for web applications to be able to scale during run time to match the workload variations.

Scaling a web application requires careful matching of system resources and the runtime workload. Such matching is challenging because web applications usually have strict Quality of Service (QoS) requirement such as bounded response time. Since the workload for web application has large fluctuation in both micro-level (within minutes) and macro-level (hours to days), dynamically matching system resources and the runtime workload in order to always satisfy the QoS requirement is very

- *Q. Wang, H. Chen, S. Zhang are with the Division of Computer Science and Engineering, Louisiana State University, Baton Rouge,LA, 70803. L. Hu is with Computing and Information Sciences, Florida International University. B. Palanisamy is with the School of Information Sciences, University of Pittsburgh, Pittsburgh, PA. E-mail: {qwang26, hchen46, szhan45}@lsu.edu, lhu@cs.fiu.edu, and bpalan@pitt.edu. Q. Wang and H. Chen contributed equally to this work.*

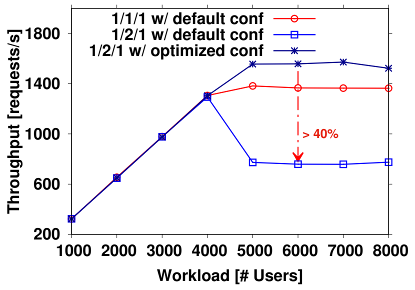


Fig. 1: System throughput comparison as a 3-tier system scales out from 1-1-1 to 1-2-1. Here 1-1-1 means one Apache web server, one Tomcat server, and one MySQL server in the system. In this experiment, Tomcat is the bottleneck server and we scale out Tomcat from 1 server to 2 as the system workload increases. Surprisingly, the maximum achievable throughput of the system decreased significantly after Tomcat scaling out.

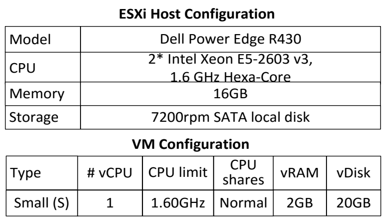
challenging. Previous work [2], [3], [4], [5], [6], [7] has proposed various scaling mechanisms by adding/removing server VMs to handle workload variations. However, these previous research efforts mainly focus on how (e.g., virtual machine live migration [3]) and when (e.g., pro-active and re-active scaling based on workload prediction [2]) to add or remove hardware resources such virtual machines to change the system capacity. Little has been discussed about how to reconfigure software components to match the hardware resource changes in the system. On the other hand, soft resources such as server threads and database connections that control the concurrency of request processing in the system have been shown to have significant impact on n-tier web application performance [8].

In this paper we show that effective scaling of an n-tier application needs intelligent coordination of both hardware and

Software Stack

Web Server	Apache 2.0.54 + tomcat-connectors-1.2.28
Application Server	Tomcat 7.0.55 + mysql-connector-java-5.1.19
Load Balancer	HAProxy 2.0
Database server	MySQL 5.0.51a
Operating system	RHEL 6.3 (kernel 2.6.32)
Hypervisor	VMware ESXi v6.0
JDK Version	Oracle JDK 1.6.0

(a) Software Stack



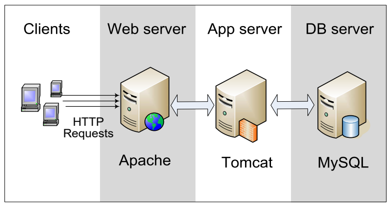
ESXi Host Configuration

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

VM Configuration

Type	# vCPU	CPU limit	CPU shares	vRAM	vDisk
Small (S)	1	1.60GHz	Normal	2GB	20GB

(b) Hardware Specification



(c) 1-1-1 Sample Topology

Fig. 2: Detailed experimental setup

soft resources scaling. Figure 1 shows one of our experimental results using a representative benchmark web application (RUBBoS) to demonstrate the importance of hardware and soft resources coordination during system scaling. After adding one more Tomcat application server VM to the original 3-tier system (one Apache web server, one Tomcat application server, and one MySQL), the maximum achievable throughout of the system unexpectedly reduced 40%. The detailed explanation of this case will be in Section 2.2. The main reason is because of the complex dependencies among the hardware and soft resources of component servers in the system; adding or removing servers in any tier of the system will change the level of the concurrent requests flowing to the downstream tiers, which may either under- or over-utilize the critical hardware resources in the downstream tiers, causing significant system performance degradation.

Concretely, we build a dynamic concurrency management (DCM) framework that takes intelligent control of soft resource allocation into the management of system scaling. DCM exploits a novel concurrency aware model that can decide a near-optimal soft resource allocation of each server in the system by combining some operational queuing laws and fine-grained monitoring data collected from each server's request processing log. We implement DCM as a two-level control framework. The first level is scaling hardware resources (e.g., VMs) of the system based on the workload variation similar as previous hardware-only scaling mechanisms. The second level is reallocating soft resources of each related server based on the concurrency-aware model recommendation after the scaling of hardware resources.

The first contribution of the paper is a sensitivity analysis of the performance impact of soft resource allocation on typical servers in an n-tier application. Through extensive benchmark experiments using realistic workload traces, we observed that the optimal soft resource allocation for different type of servers can be very different. For example, a Tomcat application server achieves the best performance when 20 threads are allocated while the optimal number is 36 for a MySQL database server in our experimental environment. We also observed that under the same hardware resource configuration a sub-optimal (but typical) allocation of threads can degrade the maximum achievable throughput of Tomcat up to 70% and 64% for MySQL (Figure 3).

The second contribution is the concurrency-aware model that determines a near-optimal soft resource allocation of each server in an n-tier system. This model takes the non-linear multi-threading overhead into account, thus the performance of each component server can be correctly characterized under high concurrency workload (Section 3). Our experimental evaluation using a representative 3-tier web application benchmark (RUBBoS) show that the optimal soft resource allocation predicted by the model actually enable the system to achieve the highest throughput compared to other typical allocation cases, validating the accuracy of the model.

The third contribution is a dynamic concurrency management (DCM) framework that exploits the concurrency-aware model to coordinate the hardware and soft resources provisioning in system scaling management (Section 4). Using six realistic bursty workload traces, our experiment results show that DCM achieves significantly better performance and higher resource efficiency than Amazon EC2-autoScale, the state-of-the-art hardware-only scale solution in a commercial cloud (Section 5).

We outline the rest of this paper as follows. Section 2 illustrates the impact of request processing concurrency on representative server performance. Section 3 introduces our concurrency-aware model. Section 4 introduces the design of our DCM framework. Section 5 shows evaluation results. Section 6 discusses related work and Section 7 concludes the paper.

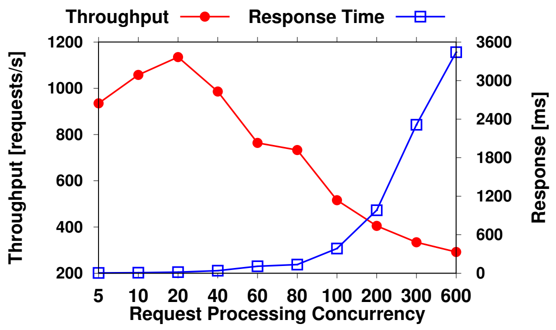
2 Backgroud and Motivation

2.1 Experiment Environment

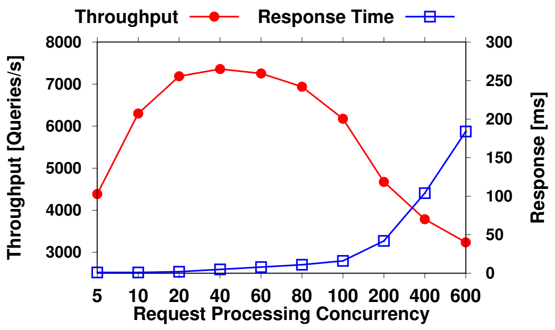
In our experiments we use a standard n-tier benchmark RUBBoS [9]. RUBBoS benchmark application is a mini-version of the popular news website Slashdot [10]. It is typically deployed as a 3-tier (web tier, application tier, and database tier) or 4-tier (with an additional load balancer for databases). The benchmark application has 24 servlets providing different web interactions. Based on the characteristics of each servlet, RUBBoS provides two workload modes: browse-only CPU intensive or read/write mix workload. We use the former mode of workload in this paper.

Figure 2 shows the experimental setup of our private VMware ESXi cluster. We adopt #W-#A-#D, a 3-digit notation to represent the number of Apache web servers, Tomcat application servers, and MySQL database servers of our benchmark application. For each #W-#A-#D, we use #W_T/#A_T/#A_C to represent three representative soft resources in the system: Apache server thread pool, Tomcat server thread pool, and Tomcat server database (DB) connection pool. They control the maximum request processing concurrency in Apache, Tomcat, and MySQL, respectively. Each servlet uses one dedicated DB connection pool in the original RUBBoS implementation. We changed the implementation to let every servlet share a global DB connection pool. The purpose is to limit precisely the number of concurrent database queries sent to the downstream MySQL. In addition, we developed monitoring tools to enable runtime monitoring and scaling of the three types of soft resources, with more details in Section 4.2.2.

We use three types of workload generators in our experiments: Jmeter [11], the original RUBBoS workload generator, and the



(a) Tomcat achieves the “best” performance when the request processing concurrency is 20.



(b) MySQL achieves high throughput when the request processing concurrency is between 20 to 80.

Fig. 3: Throughput of typical servers in an 3-tier system at increasing request processing concurrency. (a) and (b) show that improper concurrency settings in Tomcat and MySQL cause poor performance, suggesting the importance of concurrency control in the system.

revised RUBBoS workload generator. Jmeter is to generate workload with precisely controlled request concurrency [1], enabling a quantitative analysis of the impact of request processing concurrency on n-tier application performance. The original RUBBoS workload generator creates HTTP requests to interact with the benchmark application, the request rate of which follows a Poisson distribution with the mean determined by the number of concurrent users. The revised RUBBoS workload generator generates HTTP requests with realistic burstiness level based on a trace file from a production environment.

2.2 Performance Degradation with Sub-Optimal Concurrency Setting

E-commerce web applications such as Amazon.com typically process high concurrent HTTP requests from clients ranging from hundreds to thousands per second. The request processing concurrency inside the system is usually controlled by the allocation of soft resources such as worker threads or database connections of each component server. Here we use concrete experiments to show that request processing concurrency controlled by the allocation of soft resources has significant impact on the performance of typical component servers in an n-tier web application. The results help explain the unexpected performance degradation after the system scaling out as we have observed in Figure 1.

We conduct a quantitative evaluation on the performance of MySQL and Tomcat under different concurrency settings of request processing as shown in Figure 3. In this set of experiments, we use Jmeter to extract the HTTP requests recorded in a standard RUBBoS workload trace and sends the requests with precisely controlled concurrency to stress either the Tomcat (Figure 3(a)) or the MySQL (Figure 3(b)) server. For each controlled request concurrency level, we set the same number of threads in the corresponding server to avoid the queue overflow problem. In this case, the workload concurrency equals the request processing concurrency inside the server. Figure 3(a) shows the impact of request processing concurrency on the Tomcat server throughput. We can see that Tomcat achieves the highest throughput as the request processing concurrency equals 20; either lower or higher concurrency could lead to significant throughput degradation. We also observed the similar phenomenon for MySQL as shown in Figure 3(b). The only difference is that MySQL achieves high throughput at a different request processing concurrency range (between 20 to 60). Such experimental results indicate that the performance of both Tomcat and MySQL is very sensitive to the request processing concurrency.

The sensitivity of request processing concurrency in component servers also explains the unexpected performance degradation after the system scales from 1-1-1 to 1-2-1, shown in Figure 1. In that case, the 3-tier system has one Apache web server, one Tomcat server, and one MySQL server (1-1-1) at the beginning, with the default soft resource allocation 1000/100/80, which means there are 1000 Apache threads, 100 Tomcat threads, and 80 database connections. With this soft resource allocation, the maximum request processing concurrency level in MySQL is limited to 80. Our measurements show that Tomcat is the bottleneck server at the beginning, so we add one more Tomcat into the system (now becomes 1-2-1) as the workload exceeds the initial system capacity. Since we still use the default soft resource allocation for the second Tomcat, the maximum number of concurrent requests that can flow to the downstream MySQL doubles (from 80 to 160). As a result, MySQL CPU efficiency degrades significantly due to the increased request processing concurrency (see Figure 3(b)), causing unexpected system throughput drop. To truly scale the original system and fully utilize the newly added Tomcat, the database connection pool size of each Tomcat server needs to be adapted to 20. In this case, MySQL achieves the peak throughput since the maximum request processing concurrency in MySQL is limited to 40 (see Figure 3(b)).

The previous experimental results show the significant impact of soft resource allocation on the n-tier application scaling management; only scaling hardware resources without appropriate adaption of soft resource allocation could lead to significant performance degradation. Considering the common practice of system scaling in the face of naturally bursty workload for n-tier applications, smart runtime adaptation of soft resource allocations should be integrated into system scaling management.

3 Concurrency-aware Model

In this section we introduce our concurrency-aware model that determines the optimal allocation of soft resources in each component server of an n-tier system. The model is an extension of the classic queuing network model, with two additional enhancements: first, it captures the realistic request processing flow

1. Jemeter uses threads to simulate real-world users. We set zero think time between consecutive requests sent from the same thread, then we can precisely control the workload concurrency for the system as the # of Jmeter threads.

Symbol	Description
M	Number of application tiers
T_m	The mth tier in the system ($1 \leq m \leq M$)
K_m	Number of servers in tier T_m
U_m	Server utilization in tier T_m
X	Throughput of the whole system
X_m	Throughput of the mth tier
V_m	Visit ratio for tier T_m
V_b	Visit ratio for the bottleneck tier
S_m	Service time of the mth tier
S_b	Service time of the bottleneck tier
$S_b^\star$	Adjusted service time of the bottleneck tier
N_m	Number of threads in tier T_m
N_b	Number of threads in bottleneck tier b
α, β, γ	Correlation coefficients

TABLE 1: Descriptions of parameters in our model

within an n-tier system, for example, the processing of one HTTP request for a Apache web server may trigger multiple sub-queries to the downstream MySQL; second, the model considers the non-trivial multithreading overhead of each server in the system when facing high concurrency workload (see Section 2.2). Our goal is to achieve the highest system throughput through optimizing soft resource allocation in each tier of the system, in case of potential hardware configuration changes due to system scaling.

3.1 Concurrency-Aware Queue Model

Assume there are M tiers in an n-tier application, where each tier is denoted by T_1,...,T_M. We use K_m to represent the number of servers in tier T_m, where $1 \leq m \leq M$. To simplify the analysis, we start with one server in each tier at the beginning, so K_m equals 1. Assume U_m denotes the server utilization in tier T_m, then based on the Utilization Law and Forced Flow Law [12], for each tier we have the following equations:

$$U_m = X_m * S_m \quad \text{and} \quad X_m = X * V_m \tag{1}$$

In the above equations, X_m and S_m represent the throughput and average request service time of the tier T_m, respectively. X means the overall system throughput and V_m means the visit ratio of T_m. The visit ratio V_m depends on the workload characteristics. For example, Figure 4 shows that one sample HTTP arriving to Apache triggers one AJP request to Tomcat, which in turn issues two database queries to the downstream MySQL. In this case, the visit ratio $V_2 = 1$ and $V_3 = 2$. Equation 1 can be further transformed to:

$$X = \frac{U_m}{V_m * S_m} \tag{2}$$

Considering that S_m denotes the average request service time of the tier T_m, then $V_m * S_m$ means the overall service demand of an HTTP request for the tier T_m. Since we only have one server in each tier at the beginning, the bottleneck tier of the system can be easily figured out: it is the tier that has the highest service demand $\max\limits_{1 \leq m \leq M}(V_m * S_m)$. Let T_b be the bottleneck tier, then we get the maximum system throughput X_{max} when $U_b = 1$, indicating 100% utilization of the bottleneck tier resource. X_{max} can be expressed in the following equation:

$$X_{max} = \frac{1}{V_b * S_b} \tag{3}$$

In reality we may have K_b servers in the bottleneck tier, thus we transform the above equation to

$$X_{max} = \frac{\gamma * K_b}{V_b * S_b} \tag{4}$$

where γ is the correction parameter when multiple servers in the bottleneck tier is considered. This is because the system throughput will likely not to double if we double the number of servers in the bottleneck tier (e.g., from one server to two) because of many practical factors, including the load imbalance problem among servers in the bottleneck tier or the resource sharing of the downstream tiers.

When a system with a fixed configuration has stable workload characteristics, both K_b and V_b are determined. So based on Equation 4, we can predict X_{max} once the bottlneck tier service time S_b is determined. However, determining the real service time of each tier is non-trivial, especially when one HTTP request triggers several interactions between different components in the system. For example, Figure 4 shows the processing of one request in Tomcat involving two waiting periods for the response from the downstream MySQL, which breaks the real computation time for the request into three segments (the three blue boxes under Tomcat). The situation becomes more complicated when multithreading is involved. In a multi-threaded case, multiple threads may contend for shared resources, in which the original service time in a single-threaded environment may change because of the impact of resource contention as discussed in many previous work [13], [14], [15]. So to estimate the average service time of each tier in a realistic multithreading environment, we use a different method as we will discuss in the next subsection.

3.2 Service Time in Multithreading Environment

Assume the bottleneck tier has a single-threaded server and its service time is S_b^0. Now we want to know the impact of multithreading on the service time S_b^0. There are two factors that may delay the original single-threaded service time: thread contention and crosstalk penalty. The second factor is also known as coherence or consistency penalty.

Thread contention is caused by the sharing of limited software (e.g., lock) or hardware resources (e.g., CPU core) during multithreading. Thread contention may lead to the switch between threads on fine-grained time granularity (e.g, each clock), causing the interleaved execution of instructions from multiple threads. The most common switch pattern between threads is round-robin. As a result, we can model the delay caused by thread contention as linear growth with the number of threads.

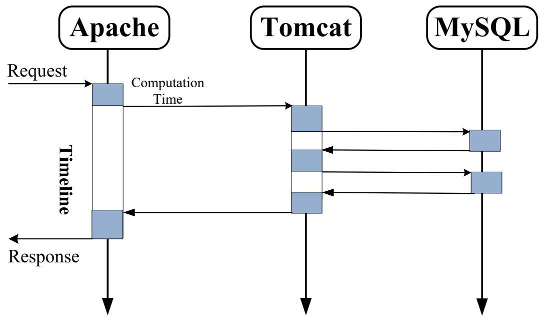


Fig. 4: Illustration of inter-tier interactions for processing one HTTP request in a 3-tier system.

Crosstalk penalty is because of the coherence or consistency requirements in a multithreading or a multi-processor environment. Hennessy and Patterson [16] (Chapter 5.2 in the Fifth Edition) has a detailed discussion about the crosstalk penalty in a centralized shared-memory architecture. In a multi-processor environment, if each processor has a thread operate on a shared variable, the crosstalk penalty is related to the number of processors in the machine because of the cache coherence requirement. Additionally, from the soft resource perspective, if each of N threads wants to obtain a mutex lock, the worst case involves N*(N-1) notification messages, since each time one thread needs to notify the other N-1 threads when it releases the mutex lock. This means that the crosstalk penalty may grow quadratically as the number of threads increases.

Based on the above two factors that affect S_b^0 under N_b threads, we can derive the adjusted service time as follows:

$$S_b^* = S_b^0 + \alpha_b(N_b - 1) + \beta_b N_b(N_b - 1) \tag{5}$$

where α_b, β_b are coefficients that depend on many factors such as hardware specification and workload characteristics. This equation also shows that when S_b^* reverses back to S_b^0 when $N_b = 1$, the single-threaded case. We note that Gunther et al. [17] have provided a formal proof of a different form of Equation 5 when they derive their Universal Scaling Law (USL). Interested readers can refer their paper for more details.

The above analysis shows that multithreading may cause longer delay for individual request processing because of thread contention and crosstalk penalty, however, multithreading enables full utilization of CPU resource and increases system throughput by taking advantage of the pipeline design of modern CPU architecture. As Figure 5 illustrates, the N_b threads use CPU cycles in an interleaved manner; one thread finishing one request takes S_b^* time, which also includes the waiting time for other (N_b - 1) threads. Assuming each of N_b threads shares the CPU fairly, then the adjusted average service time for each thread is:

$$S_b = \frac{S_b^*}{N_b} = \frac{S_b^0 + \alpha_b(N_b - 1) + \beta_b N_b(N_b - 1)}{N_b} \tag{6}$$

By combining Equestion 4 and 6, we can derive the system's maximum throughput as a function of the concurrency in the bottleneck tier as shown below:

$$X_{max} = \frac{\gamma * K_b * N_b}{S_b^0 + \alpha_b(N_b - 1) + \beta_b N_b(N_b - 1)} \tag{7}$$

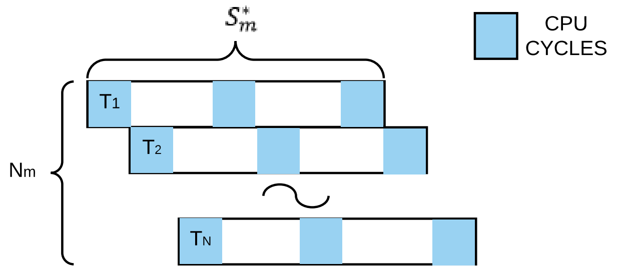


Fig. 5: Pipeline processing of requests with multi-threads

3.3 System Throughput Maximization

Equation 4 shows that to get the maximum throughput of the whole system, the bottleneck tier service time S_b needs to be minimized. Thus we transform Equation 6 as follows:

$$\begin{aligned} S_b &= \frac{S_b^0 - \alpha_b}{N_b} + \beta_b N_b + (\alpha_b - \beta_b) \\ &\geq 2\sqrt{(S_b^0 - \alpha_b)\beta_b} + (\alpha_b - \beta_b) \end{aligned}$$

We get the minimum S_b when $N_b = \sqrt{\frac{S_b^0 - \alpha_b}{\beta_b}}$. Then we take Min($S_b$) back to Equation 4, and get the maximum system throughput as follows:

$$Max(X_{max}) = \frac{\gamma * K_b}{V_b(2\sqrt{(S_b^0 - \alpha_b)\beta_b} + \alpha_b - \beta_b)} \tag{8}$$

The above derivation process of the model shows that to maximize the whole system throughput, we need to set the number of threads in the bottleneck tier server to be N_b. The value of N_b depends on the parameters S_b^0, α_b, and β_b. S_b^0 can be measured through system profiling. The other two parameters can be determined via regression analysis based on the online measurement of system throughput and the allocation of threads in each server of the bottleneck tier. Section 3.4 will show concrete parameter training and validation for different bottleneck servers in the system.

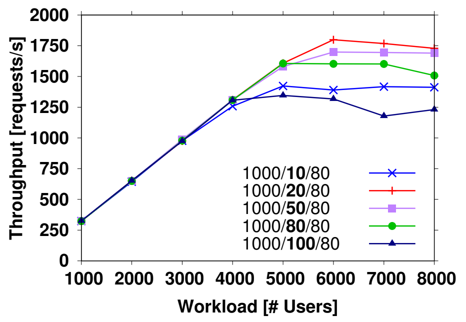
We note that setting the optimal concurrency N_b of the servers in the bottleneck tier does not guarantee the maximum system throughput; we also need to set proper number of threads and connections in the upstream tiers in order to allow enough concurrent requests flowing to the bottleneck tier and fully utilize the bottleneck resource. Interested readers can refer to our previous paper [18] that characterizes the relationship between soft resource allocations between different tiers.

3.4 Model Training and Validation

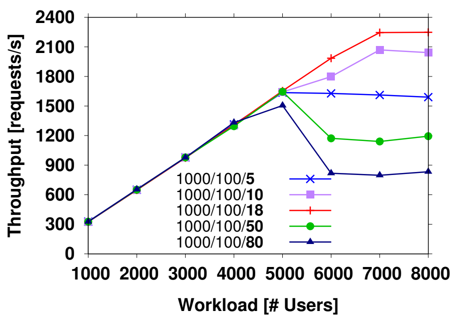
To fast train the parameters of the model and determine the optimal soft resource allocation for each component server, we use Jmeter to generate workload (i.e., HTTP requests) with different concurrency levels extracted from the real RUBBoS workload trace. Because we set the user think time between consecutive HTTP requests from the same user (thread) to be zero, we can control the exact request concurrency to the target system by specifying the number of users (threads) in Jmeter. Then we return back to the original RUBBoS workload generator in the validation phase since it simulates the realistic production workloads. Our goal is to validate whether the model recommended optimal soft resource allocation could achieve the maximum system throughput under realistic workload scenarios.

Model training for Tomcat: The concurrency-aware model is to build the correlation between the request processing concurrency in Tomcat and the system throughput in order to determine the optimal concurrency setting in Tomcat. To build such a model, we use the 1-1-1 configuration (with the default soft resource allocation 1000/100/80) because Tomcat is the bottleneck tier of the system. Then we increase the workload concurrency for the system from 1 to 100 and record the system throughput (see Figure 3(a)). We take the <*concurrency, throughput*> pairs [2]

2. *Concurrency* and *throughput* correspond to N_b and X_{max}.



(a) Model validation of Tomcat threads in the 1-1-1 case



(b) Model validation of Tomcat DB connections in the 1-2-1 case

Fig. 6: Model validation for Tomcat and MySQL using realistic system configuration. (a) and (b) show that the model recommended optimal concurrency setting (20 for Tomcat while 36 for MySQL according to Table 2) indeed outperforms other four representative allocations. We note that the 1-2-1 case has two Tomcat servers, thus 1000/100/**18** can make sure the optimal concurrency (36) in MySQL.

as the input to the model as shown in Equation 7. By applying the Least-Square Fitting analysis we estimate the values of the parameters of the model and also the optimal concurrency setting ($N_b = 20$) in Tomcat as shown in Table 2. The statistical R-Squared value is 0.960 according to our additional measurement data, indicating high accuracy of the model in predicting system throughput under different Tomcat concurrency.

We further ran experiments using the original RUBBoS workload to validate the model generality. The RUBBoS workload simulates the realistic workload scenario in which a user within the same session sends every follow-up HTTP request after a certain amount of "think" time. Figure 6(a) shows the experimental results for the 1-1-1 case with five representative soft resource allocations. 1000/**20**/80 is recommended by the model. This figure shows that the system with the "optimal" soft resource allocation indeed outperforms the other cases. For example, the "optimal" allocation case outperforms the default configuration case (100 Tomcat threads) 30% in throughput.

Model training for MySQL: To train the model for MySQL, we scale the system from the previous 1-1-1 configuration to 1-2-1 since MySQL is the bottleneck server under the new configuration. Then we conduct the similar experiments of model training for MySQL as for Tomcat previously. The estimated values of the model parameters are shown in the third column of Table 2. After resolving the model we conclude that the system achieves the maximum throughput when MySQL threads allocation $N_m = 36$ (Figure 3(b) shows partial <*concurrency, throughput*> pairs). We note that in our experiments we use the DB connection pool size in Tomcat to control the request processing concurrency in MySQL and there are two Tomcat servers in the system, thus the optimal allocation of DB connections in each Tomcat should be 18. We further conduct experiments using the more realistic RUBBoS workload to validate the generality of the model. Figure 6(b) shows that the model recommended allocation (1000-100-**18**) indeed outperforms the other four representative cases, including the default case with 80 DB connections. We note that 100 threads in Tomcat is chosen because we want avoid Tomcat thread pool being the bottleneck that limits the number of concurrent requests flowing to the downstream MySQL.

Model retraining to keep prediction accuracy: In the above experiments we have validated the accuracy of our model in realistic workload scenarios. We need to point out that the model is based on two assumptions. First, the characteristics of the

TABLE 2: Model training and prediction result. S_b^0 is measured through system profiling.

Parameter	Tomcat Model	MySQL Model
S_b^0	2.84e-02	7.19e-03
α_b	9.87e-03	5.04e-03
β_b	4.54e-05	1.65e-06
γ	11.03	4.45
R^2	0.960	0.97
N_b	**20**	**36**
X_{max}	946	865

workload (e.g., read/write ratio) in the training phase and in the production phase (i.e., the realistic workload scenarios) keep the same. Second, the new servers added into the bottleneck tier are homogeneous to the other servers in the same tier. This is because according to Equation 8, both the maximum achievable throughput and the optimal concurrency setting of a server N_b is related to the basic service time S_b^0, which depends on two factors: the workload characteristics and the hardware provisioning (e.g., # of CPU cores and frequency) of the server. S_b^0 will change if any of these two factors changes, thus the optimal concurrency setting N_b of the server predicted by the model will change and no longer be the optimal in the production phase. To always keep the model accuracy, we need to retrain the model of each server based on the online monitoring data collected from real production environment from time to time, assuming that the workload characteristics may change over time or the system scales using heterogeneous hardware provisioning.

4 Dynamic Concurrency Management Design and Implementation

The previous section describes a concurrency-aware model for the optimal concurrency setting of the bottleneck tier in the system based on measurement data. Since system scaling in/out potentially changes the request processing concurrency in the system, to always maintain the high performance of the system, we describe a dynamic concurrency management (DCM) framework which is to dynamically adjust soft resources allocation in related servers based on the model prediction.

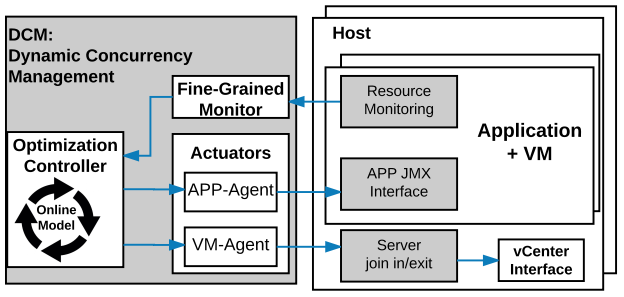


Fig. 7: The DCM framework

Figure 7 shows the DCM framework. It includes three key components: Fine-Grained Resource Monitor, Optimization Controller, and Actuators.

Fine-Grained Resource Monitor: Each VM installs a monitoring agent to collect both the application-level metrics (e.g., # of active threads, average server response time and throughput) and system-level metrics (e.g., CPU, Memory, network I/O). Then monitoring agents send the measured data at every second back to a storage server (Kafka [19]). The controller will consume these data for runtime performance analysis. The purpose of Kafka is to serve as an intermediate storage server to coordinate the distributed monitoring agents that produce data and the controller that consume data since the controller may need to operate on collective data over a period of time (e.g., 1 minute).

Optimization Controller: The controller makes adaptation decisions based on the analysis of the data from Kafka and the concurrency-aware model as we present in Section 3. Two decisions need to be made at the moment of burst workload: VM-level scaling and soft resources re-allocation. VM-level scaling is to decide when to launch new VMs to improve the bottleneck tier performance or turn off idle ones to avoid wasting computing resources. Soft resources re-allocation is to make them best suit the concurrency requirement of servers in the system after the VM-level scaling finishes. In our current implementation, the controller adopts the resource-usage driven approach in the VM-level scaling, meaning that the controller will trigger the execution of Actuators once the resource usage of any tier exceeds a predefined threshold (e.g., 80%).

Actuators: The DCM has two actuators. the VM-agent actuator is to start or turn off VMs in a specific tier. The APP-agent actuator is to re-allocate soft resources in the system based on the concurrency-aware model recommendation. Usually the APP-agent actuator follows right after the VM-agent actuator.

In the following section we outline the control algorithm of how the above three components interact with each other for intelligent system scaling management.

4.1 Dynamic Concurrency Management Algorithm

Our algorithm makes the following three assumptions:

1) There is only one bottleneck tier at a time in the system.
2) Our monitoring tools are able to identify the bottleneck hardware resource as system performance deteriorates.
3) We are able to dynamically adjust the soft resource allocation in each server during run time.

The first assumption is to make sure that the system does not encounter the complex multi-bottleneck scenario [20], [21]. A multi-bottleneck scenario refers to the case where the bottleneck shifts rapidly among different system components because of complex resource dependencies in the system. In such a case, the system performance may deteriorate while the average utilization of each component is far from saturation, which disables the triggering conditions (e.g.,CPU utilization higher than 80%) of the control framework. Handling complex multi-bottleneck cases still remains a significant challenge and needs our further research. The second assumption assumes that we have proper monitoring tools such as *collectl* and *sysstat* to enable bottleneck detection. The third assumption assumes that we have management tools which are able to scale soft resources of each component server on the fly. While there are many existing monitoring tools to satisfy the second assumption, we implement our own management tools for soft resource scaling, with more details in Section 4.2.2.

Algorithm 1: Pseudo-code for DCM scaling control

```
1  procedure DCMScalingControl
2  $scaleOutStep$ = 1, $scaleInStep$ = -1;
3  $slowTurnOffFactor$ = 3, $SystemRunning$ = true;
4  while ($SystemRunning$) do
5      $/*\ Record\ hardware\ resources\ beyond\ util.\ threshold\ */$
6      ($R_h$,$R_l$) = ResourceMonitor();
7      if ($R_h \neq \phi$) then
8          $/*\ hardware\ resource\ util.\ exceeds\ upperbound\ */$
9          $bottleneckTier$ = $R_h \to k$;
10         VMScale($bottleneckTier$, $scaleOutStep$);
11         SoftResourceScale($bottleneckTier$, $scaleOutStep$);
12     else if ($R_l \neq \phi$ && $counter > slowTurnOffFactor$) then
13         $/*\ hardware\ resource\ util.\ cont.\ below\ lowerbound\ */$
14         $counter$ = 0;
15         $scaleInTier$ = $R_l \to k$;
16         if ($nTier[scaleInTier] > abs(scaleInStep)$) then
17             VMScale($scaleInTier$, $scaleInStep$);
18             SoftResourceScale($scaleInTier$, $scaleInStep$);
19     else if ($R_l \neq \phi$ && $counter < slowTurnOffFactor$) then
20         $counter$++;
21     else
22         $counter$ = 0;
23     end
24 end
25 procedure VMScale ($scaleTier$, $scaleStep$)
26 if ($scaleStep > 0$) then
27     TurnOnVMs($scaleTier$, $scaleStep$);
28     $nTier[scaleTier]$ = $nTier[scaleTier]$ + $scaleStep$;
29 else if ($scaleStep < 0$ && ($nTier[scaleTier]$ + $scaleStep$) > 0)
     then
30     TurnOffVMs($scaleTier$, abs($scaleStep$));
31     $nTier[scaleTier]$ = $nTier[scaleTier]$ - abs($scaleStep$);
32 procedure SoftResourceScale ($scaleTier$, $scaleStep$)
33 if ($scaleStep > 0$) then
34     $/*get\ optimal\ concurrency\ of\ each\ downstream\ server*/$
35     $optAlloc$ = ModelPredict($scaleTier$ + 1);
36     soft[$scaleTier$] = ($optAlloc$ * $nTier[scaleTier + 1]$)
37                         /($nTier[scaleTier]$ + $scaleStep$);
38     AppActuator($scaleTier$, soft[$scaleTier$]);
39 else if ($scaleStep < 0$) then
40     $/*\ get\ optimal\ concurrency\ of\ scaleTier\ server\ */$
41     $optAlloc$ = ModelPredict($scaleTier$);
42     soft[$scaleTier$] = $optAlloc$;
43     AppActuator($scaleTier$, soft[$scaleTier$]);
```

Algorithm 1 shows the pseudo-code for the interaction of the three components in our control framework. We explain the key procedures in more detail in the following:

DCM Scaling Control. This procedure details the control logic of the Optimization Controller. The controller exploits resource monitors to measure the runtime system status such as CPU utilization and application performance metrics and make scaling decisions. We define two resource utilization thresholds: upper bound for scaling-out and lower bound for scaling-in. For each control period, the resource with average utilization higher than the upper bound (e.g., 80%) is recorded in R_h while the one lower than the lower bound (e.g., 40%) is recorded in R_l (line 6).

To make the system performance stable under bursty workload, the controller adopts the "quick start but slow turn off" scaling policy learned from the AutoScale work of Gandhi et al. [2]. Concretely, if the utilization of any concerned resource during one control period exceeds the pre-defined upper bound, the controller will ask the VM-agent actuator to launch new VM(s), which will spend certain amount of preparation time (e.g., 15 seconds) before joining the system and ready to serve requests. On the other hand, the controller will turn off VMs where the utilization of all the concerned resources is below the lower bound continuously for three control periods (line 12).

Virtual Machine Scaling. This procedure is to turn on/off virtual machines similar as all the other scaling frameworks in the cloud (e.g., Amazon EC2 AutoScale). While turning on VMs is relatively easy, turning off VMs needs to check whether the number of VMs to turn off (requested by controller) is less than the number of running servers (line 30).

Soft Resource Scaling. This procedure is to scale the soft resource allocation of the system based on the concurrency-aware model after the VM level scaling. Theoretically, the optimal concurrency setting in each VM does not change with the number of VMs, however, the actual request processing concurrency in a server is also affected by the size of connection pools of the servers upstream tier, which may change when the server's upstream tier scales out or in. In this procedure, we simply make the size of connection pools in one tier the same as that of the thread pools in its successive downstream tier after the VM-level scaling. Nevertheless, this procedure shows that the soft resource scaling is different between VM scaling out and in. For VM scaling out, we need to consider the impact of the increased request concurrency (due to newly added VMs) on the performance of the downstream tier servers. Assuming that the optimal concurrency (based on our concurrency-aware model) for each downstream server is $optAlloc$ and the number of downstream servers is $nTier[scaleTier + 1]$, so the optimal total concurrency of the downstream tier is $optAlloc * nTier[scaleTier+1]$. Thus the new allocation of soft resources (connection pool size) of each server in the $scaleTier$ tier is shown in line 38. On the other hand, in a VM scaling-in case, the number of VMs in the $scaleTier$ tier is reduced while the maximum concurrent requests from its upstream tier keep the same. To avoid high overhead caused by high concurrent requests from its upstream tier, the soft resource allocation (thread pool size) of the $scaleTier$ tier needs to be re-adjusted based on the concurrency-aware model recommendation (line 44).

4.2 Implementation Details

4.2.1 VM-Agent for VM-level Scaling

Launching or turning off VMs is easy in the cloud because the underlying hypervisor provides corresponding APIs that can be called remotely. The complexity of VM level scaling comes from the servers that run inside VMs. For example, it is relatively easy to add VMs that run stateless servers (e.g., Apache web servers) because they can serve new requests seamlessly right after they join the system. However, adding VMs that run stateful servers (e.g., database servers) is non-trivial because they need to resolve data or state consistency problem, for example, a newly added database server may need to synchronize with other running databases in the system, thus may require more preparation time to be ready to serve new requests. We set the preparation period of each VM to be 15 seconds after VM-agent actuator launches the VM, which is enough for the VM to be ready in our benchmark experiments. More preparation time may be needed in real production environment. We also use HAproxy [22] as a load balancer to dynamically balance workload among servers after the system scaling, where we adopt the least pending request (LPR) scheduling policy to dispatch requests to downstream servers.

4.2.2 APP-Agent for Soft Resource Re-Allocation

Once the VM-level scaling is done, we use APP-agent to control the request processing concurrency in each component server through re-allocating soft resources in the system. Two approaches exist to limit the request processing concurrency level of a server: adjusting the servers thread pool (STP) size or controlling the upstream tier's connection pool size. The second one is possible because the connection pool size in the upstream tier can limit the maximum number of concurrent requests flowing to the server. We use the first approach (i.e., adjusting the STP size) to control the request processing concurrency in Tomcat because Tomcat may directly serve HTTP requests from clients, thus no connection pool from upstream tiers to control with. On the other hand, we use the second approach to control the request processing concurrency in MySQL. This is because we are able to control the DB connection pool size of Tomcat, which is the direct upstream tier of MySQL.

The runtime adjustment of the STP size of Tomcat is supported by the latest Tomcat implementation. Tomcat registers STP as an MBean in the hosting Java Virtual Machine (JVM), which allows a remote Java program (e.g., our APP-agent) to fetch or change the STP size through remote method invocation (RMI). However, the DBConnP parameter is not included in Tomcat server MBeans. We need to find a way to expose the capabilities of managing this parameter dynamically.

There are two methods to manage the DBConnP size in Tomcat. The first method is to dynamically change and load the Tomcat JDBC connection pool configuration file, however, this method requires change of the DB connection pool implementation of the original application. The second method is to implement our own MBean exposing the management interface of the DBConnP parameter, which is similar to the management of STP in Tomcat. Since the second method is less intrusive and can be easily applied to other Java-based servers, we choose the second method to implement the management module of DBConnP in Tomcat as part of our APP-agent component.

5 Experimental Evaluation

Here we evaluate how effectively does DCM perform when comparing to other state-of-the-art system scaling solutions under six realistic bursty workload scenarios. Concretely, we compare the performance of a 3-tier application equipped with two system scaling management frameworks: DCM and the hardware-only scaling framework "EC2-AutoScale" [23] provided by Amazon AWS. The latter one has been widely used in academic research [24], [25], [26] and industry practices. We will show that DCM outperforms "EC2-AutoScale" in both the system tail

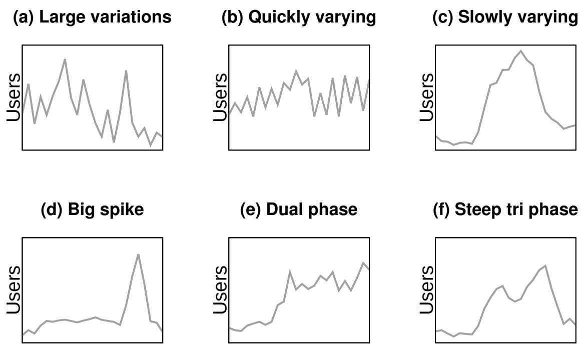


Fig. 8: Workload Traces We Use for Experiments

latency and throughput because of intelligent control of request processing concurrency during the system scaling process.

5.1 DCM Evaluation with Concurrency-Aware Models

We have implemented both DCM and EC2-AutoScale in our VMware ESXi 6.0 cluster environment. With EC2-AutoScale, customers can set threshold values (usually the CPU utilization) to dynamically add or remove VMs from an Auto Scaling group, which specifies the type and the upper bound of VMs to scale [27]. Amazon CloudWatch is used to monitor resource utilization. Concretely, EC2-AutoScale uses Amazon CloudWatch to monitor resource utilization and trigger the scaling activities once the monitored resources exceeds the predefined threshold. We set the control period for both controllers to be 15-second, which has been used in other state-of-the-art control policies [14], [28]. To avoid performance instability caused by bursty workload, we also adopt the "quick start but slow turn off" VM scaling policy learned from Gandhi's et al. AutoScale work [2], as described in more detail in Section 4.1.

We evaluate the effectiveness of the two controllers using six realistic workload traces (Figure 8) collected from the real-world production systems [29], [30]. These traces are categorized by Gandhi et al. in their AutoScale paper [2]. To fit the capacity of our experiment environment, we scale these workload traces such that the maximum number of concurrent users is 7500, and the duration of each trace is 12 minutes.

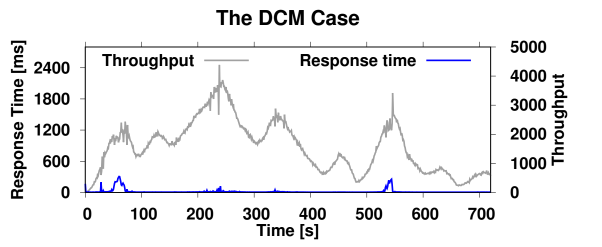
Figure 9 shows the timeline comparison between the DCM and the EC2-AutoScale cases under the same "Large variations" workload (see Figure 8(a)). The four subfigures in the left column show the DCM case and those in the right column show the EC2-AutoScale case. In both cases the system has the same initial hardware configuration 1-1-1, with the default soft resource allocations 1000/100/80. Comparing Figure 9(a) and 9(b), the DCM case shows relatively stable performance all the time while the EC2-AutoScale case has three obvious performance deterioration periods (50s∼90s, 227s∼259s, and 530s∼560s). The interesting observation is that all the three periods of performance degradation are the periods when the bottleneck tier of the system is about to scale (see Figure 9(d) 9(f)).

Taking the first period 50s∼90s in Figure 9(b) for example, EC2-AutoScale presents a large response time spike and significant throughput drop. During this period, we observe that the original one Tomcat server scales to two at 67s because the Tomcat CPU utilization exceeds the scaling threshold (see Figure 9(d)). We note that the system performance already starts to deteriorate before the second Tomcat adding in. This is because the scale-out activity is triggered only after the 15-second control period. Interestingly, the system performance degrades even further when the second Tomcat adds into the system. This is because of the increased request processing concurrency in MySQL after the Tomcat scaling out. Once the second Tomcat adds in, MySQL becomes the new bottleneck tier. Due to the newly added Tomcat, the Tomcat tier now is able to send doubled concurrent requests to the downstream MySQL (from the default 80 DB connection pool size to 160) (see Figure 9(h)). High concurrent requests in MySQL cause low efficiency of MySQL CPU (see Figure 3(b)) and thus low Queries-Per-Second (QPS, throughput of MySQL) as shown in Figure 9(h). The system performance eventually returns to normal after the second MySQL instance added to the system at time mark 90s as shown in Figure 9(f). The second performance deterioration during the period 227s∼259s is similar as the first one when the third Tomcat and MySQL are added to the system due to the continual increase of workload. The third performance deterioration period 530s∼560s is more interesting. The MySQL tier scales in from two instances to one at 528s due to the decreased workload. Then in the next control period high workload suddenly floods to the MySQL tier, and the only left-over MySQL instance encounters high request processing concurrency (160), causing low QPS (see Figure 9(h)).

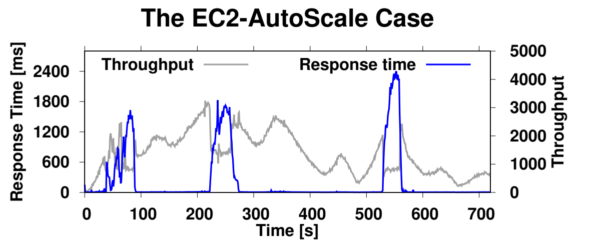
On the other hand, there is only moderate performance degradation in the DCM case during the three periods mentioned above under the same workload (see Figure 9(a)). This is because DCM dynamically reallocates soft resources in both Tomcat and MySQL to the "optimal" level based on the model prediction, thus both of them perform efficiently during the temporarily overloaded periods, thus achieves much more stable performance compared to the EC2-AutoScale case.

Readers may find out that the fundamental problem of the performance deterioration in the EC2-AutoScaling case is because of the scaling-out activities lagging of the workload increase, which leads to temporary high concurrency of the servers with degraded efficiency in the bottleneck tier. A simple solution is to reduce the control period to let the system respond fast, however, too small a control period will make the system unstable under bursty workload [31]. Even if we reduce the control period, the preparation period for a newly added VM taking effect still takes time. For example, Gandhi et al. report 30s∼1min for launching a KVM-based VM [32]. In our experimental environment, a ESXi host needs 15s to launch a VM. Some more advanced approach may scale out the system proactively based on the prediction of workload [3], [28], however, predicting n-tier application workload such as e-commerce is a well-known research challenge because of the bursty nature of the workload (e.g., Slashdot effect [10]). So temporary overloading of the system is unavoidable in practice and DCM can help stabilizing system performance during the temporary overloading periods.

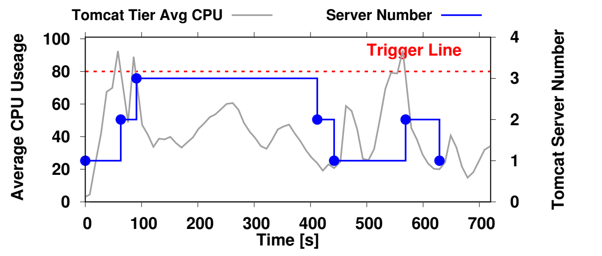
Figure 9 also shows that DCM achieves higher resource efficiency than EC2-AutoScale because DCM achieves better performance while using the same (if no less) amount of hardware resources. For example, both the Tomcat and MySQL tier in DCM and EC2-AutoScale scales up to 3 server instances during the 700 seconds experimental period. The fundamental reason is because DCM is able to dynamically adjust the software resource allocations of the servers in the system to more efficiently utilize the underlying hardware resources (e.g., CPU).



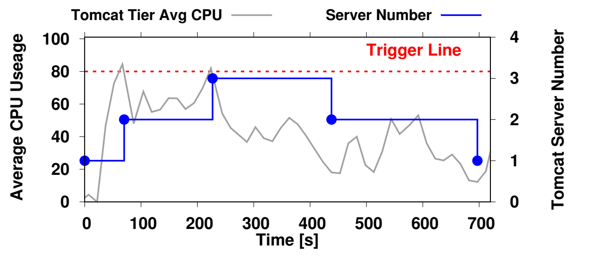
(a) System response time is low and stable under the "Large Variation" workload. System throughput matches the workload variation (Figure 8(a)), suggesting no throughput loss.



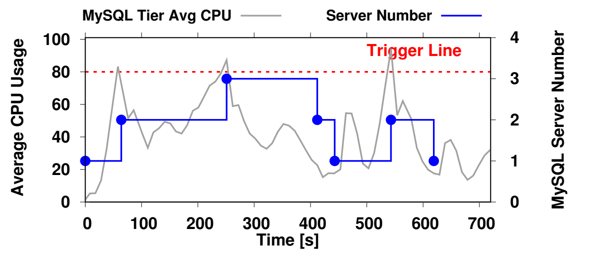
(b) System encounters frequent response time spikes under the same workload trace as in (a). System throughput drops during each response time spike period at 70s, 230s, and 530s.



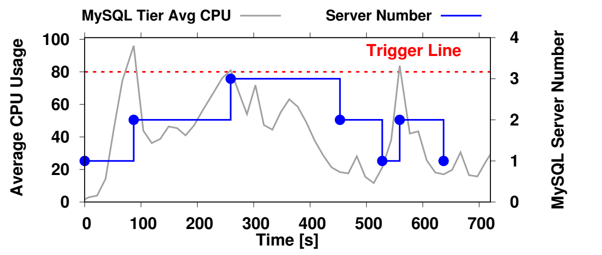
(c) The Tomcat tier scales out/in along with the CPU util. variation. The scaling-out actions at 80s, 100s, and 560s cause only minor variation of system response time as shown in (a).



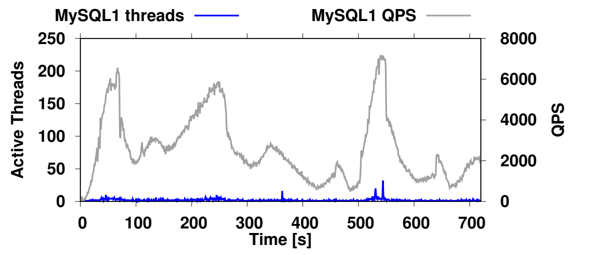
(d) In the EC2-AutoScale case, Tomcat scaling-out actions at 80s and 230s coincide with response time spike periods and throughput drops as shown in (b).



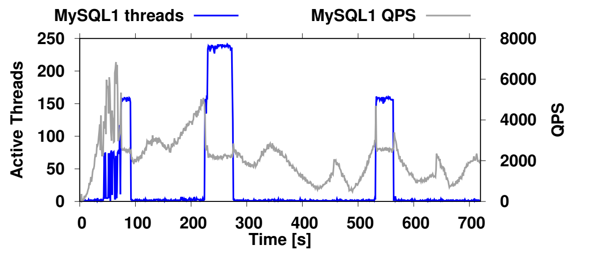
(e) The MySQL tier scales out/in along with the CPU util. variation. MySQL scaling-out actions at 80s, 230s, and 540s coincide with minor response time variation as shown in (a).



(f) In the EC2-AutoScale case, MySQL scaling-out actions at 90s, 250s, and 550s coincide with response time spike periods and throughput drops as shown in (b).



(g) Runtime request processing concurrency and Queries-Per-Second (QPS) of the first MySQL. MySQL concurrency limit is controlled to the "optimal" level at the scale-out periods 80s, 230s, and 540s (see (e)), thus produces high QPS.

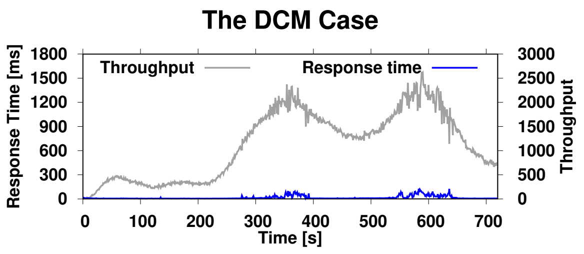


(h) MySQL runtime concurrency is high at the scaling-out periods 90s, 250s, and 550s. Suboptimal high concurrency in MySQL leads to low QPS, which in turn causes response time spikes and throughput drops as shown in (b).

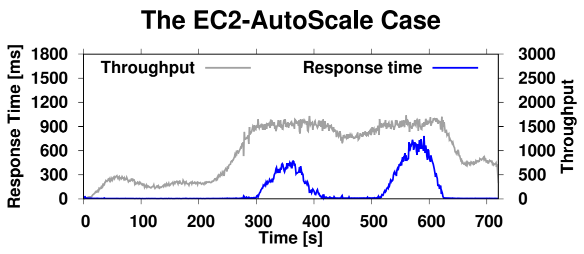
Fig. 9: Performance degradation of EC2-AutoScale compared to DCM under the same "Large Variation" workload. The left side figures are for DCM while the right side are for EC2-AutoScale. The system in both cases starts with the 1-1-1 configuration and the default 1000/100/80 soft resource allocation, however, DCM outperforms the EC2-AutoScale case once system scaling actions occur.

5.2 Performance Comparison Under Other Traces

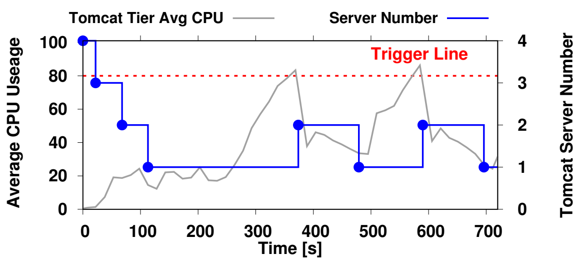
The experiments in Figure 9 are conducted when the target system starts with the default configuration 1000/100/80, showing that DCM outperforms EC2-AutoScale significantly. What if the system starts with initially optimal concurrency setting? Here we show that, static concurrency setting, even optimal at the beginning, may cause significant performance degradation when the target system scales to a different size. We show this case by conducting the experiments using the "Steep tri phase" workload trace, where the system starts with initially optimal allocation of software resources in both DCM and EC2-AutoScale. We changed the initial system hardware configuration from 1-1-1 to 1-4-1, meaning that the system initially consists of one Apache web server, four Tomcat servers, and one MySQL database server. Such initial configuration makes sense because the number of application servers (Tomcat here) is usually larger than that of the



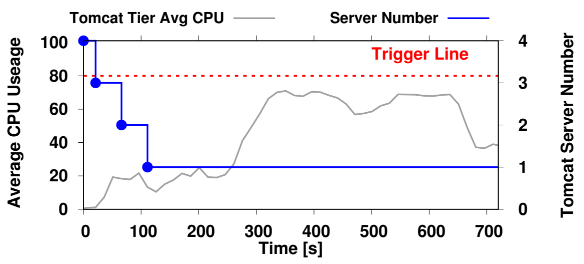
(a) System achieves stable response time under the “Steep Tri Phase” workload (see Figure 8(f)). System throughput matches the workload variation, suggesting no throughput loss.



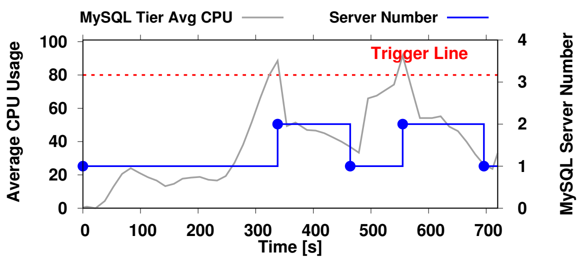
(b) System encounters response time spikes under the same bursty workload trace as in (a). System throughput also drops significantly during the response time spikes at 320s and 550s.



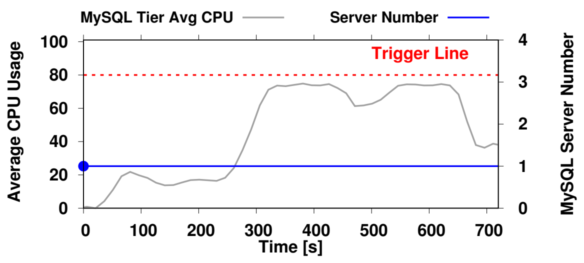
(c) The Tomcat tier scales out/in along with the CPU util. variation. Tomcat scaling-out actions at 370s and 590s cause only minor variation of system response time as shown in (a).



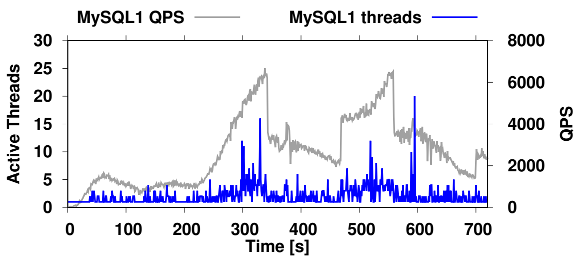
(d) In the EC2-AutoScale case, the Tomcat tier does not scale out because the CPU utilization is always below the triggering threshold (80%).



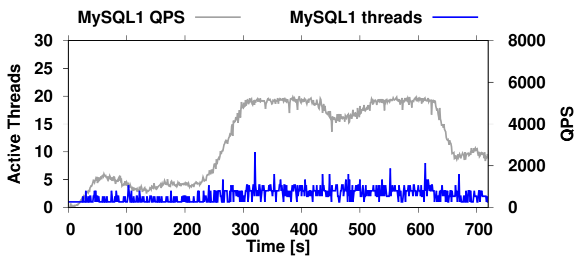
(e) The MySQL tier scales out/in along with the CPU util. variation. The MySQL tier scaling-out actions at 340s and 560s coincide with minor response time variation as shown in (a).



(f) In the EC2-AutoScale case, the MySQL tier does not scale out because the CPU utilization is always below the triggering threshold (80%).



(g) Measured runtime concurrency and Queries-Per-Second (QPS) of the first MySQL. MySQL concurrency limit is always controlled to the “optimal” level (40) during the scaling-out periods at 340s and 560s (see (c)), thus produces high QPS.



(h) In the EC2-AutoScale case, MySQL runtime concurrency is low all the time. But the QPS is lower than that of the DCM case (g) during the peak client workload period at 340s and 560s.

Fig. 10: Performance degradation of EC2-AutoScale compared to DCM under the same “Steep tri phase” workload. The 1-4-1 system initially starts with the optimal allocation of software resources in both cases, however, the DCM case shows much more stable performance than the EC2-AutoScale case by comparing (a) and (b).

database servers in a typical n-tier system since database is more likely to be the system bottleneck. Based on our concurrency-aware model, the optimal software resource allocation for the system should be 1000/20/9 (1000 threads in the Apache web server, 20 threads and 9 database connections in a Tomcat server). Such a setting is optimal because: (1) MySQL is the bottleneck server at the beginning, thus we should optimize the request processing concurrency in MySQL, which is 36 based on our model (2) there are four Tomcat servers, thus the optimal database connection pool size of each Tomcat should be 9, so the maximum number to concurrent requests that are allowed to flow to the downstream MySQL is 9*4=36. Figure 10 shows that DCM performs much better than EC2-AutoScale under the workload trace Steep tri phase. This is because the system scales-in to 1-1-1 due to the

TABLE 3: Response Time Performance Comparison Between Autoscale and DCM Under Different Traces

Percentile Response Time (ms)		Large Variation	Quick varying	Slowly varying	Big Spike	Dual Phase	Steep Tri Phase
RT_{95}	EC2-AutoScale	1027	904	1087	525	622	485
	DCM	**125**	**28**	**206**	**111**	**57**	**56**
RT_{99}	EC2-AutoScale	3566	2229.99	3228	1777	1378	1710
	DCM	**226**	**105**	**352**	**198**	**223**	**136**

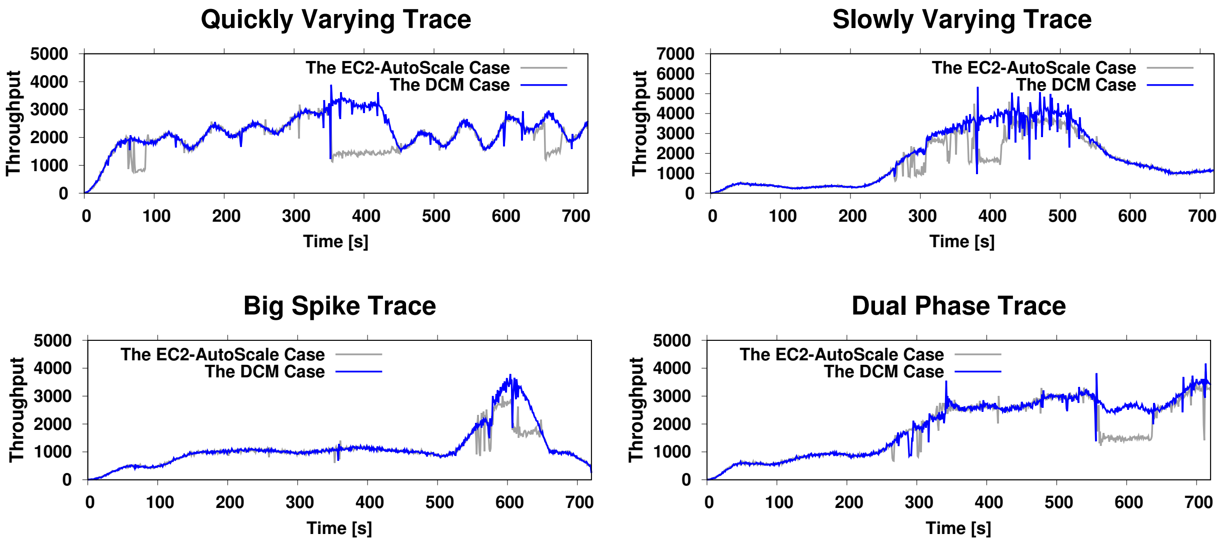


Fig. 11: Throughput comparison between DCM and EC2-AutoScale under the other four realistic bursty workload traces

low workload at the beginning. As the workload starts to increase at the period around 300s, EC2-AutoScale can not scale-out either Tomcat or MySQL because the static small database connection pool size (9) becomes the new system bottleneck, causing low utilization of Tomcat and MySQL CPU, thus can not trigger any scaling-out activities to handle the increased workload.

Figure 11 shows the throughput comparison results between DCM and EC2-AutoScale under the other four representative workload traces (see Figure 8). In most cases DCM has more stable throughput performance than EC2-AutoScale, the root cause has been explained before and we do not repeat here. The only exception is the "slowly varying" workload trace case. Although DCM does not have long sharp throughput drop as EC2-AutoScale during the system scaling out phase (between 380s to 420s), its throughput has large variation between 420s to 510s. This is because the system under control reaches the scaling out limit here. The EC2-AutoScale framework has a scaling policy in which a user has to set a scaling out limit of VMs in the Auto Scaling group [27]. To compare DCM and EC2-AutoScale in realistic scenarios, we set the scaling out limit of both the Tomcat tier and the MySQL tier to be 3 VMs each. Under the "slowly varying" workload trace case, the system already reaches the scaling out limit between 420s to 510s because of the steady high workload period (see Figure 8(c)), and the system performance starts to become unstable during this period as shown in Figure 11(b).

We further summarize the the system response time comparison results under all the workload traces in Table 3. We choose the RT_{95} and RT_{99} as the comparison metrics, which represent 95th and 99th percentile response time, respectively. The DCM case outperforms the EC2-AutoScale case uniformly under different workload traces. Even for RT_{99}, we can see that the DCM case still keeps the response time below 500ms, which (or even lower) is a common Service Level agreement (SLA) requirement for most modern e-commerce websites [28], [33], [32].

Discussion. Although DCM has been demonstrated to work effectively when scaling a small-scale 3-tier application, DCM also applies to large scale web applications where each tier has tens to hundreds of VMs. As long as the concurrency of components/tiers changes resulting from the system scaling, we need to re-adjust the concurrency setting after the system scaling in order to efficiently utilize the underlying hardware resources. For large systems, the effectiveness of DCM highly depends on the workload variation pattern; for example, if the ratio of the turned-on/off VMs over the total number of VMs in one tier is large after the system scaling, then DCM should work more effectively since the concurrency change to downstream tiers is also large. On the other hand, if the ratio of the turned-on/off VMs is small, then our DCM should be less effect since the change of request processing concurrency in the system is also small.

6 Related Work

Feedback Control Based Resource Adaption. Most previous research in this category [2], [4], [6], [32], [34], [35], [28] shares the similar goal: meet the QoS requirements such as bounded response time while minimize the cost (either operation or infrastructure). These research efforts can be further classified into two groups: reactive approaches and predictive approaches.

Reactive approaches [5], [28], [2] use either system-level (e.g., CPU utilization) or application level (e.g., system response time, queue length) feedback signals to determine when to scale the system. Due to the reactive nature, the system usually already suffered the performance damage before the newly added VMs starting to share load due to the unavoidable and sometimes very long setup time (e.g., up to minutes [32]).

Predictive approaches [3], [36], [26], [4] works well for workloads with periodic patterns. The long setup time could be avoided once the controller accurately predicts the workload and takes scaling actions upfront. However, n-tier web applications naturally have bursty workload in both macro level (see Figure 8) and micro level [31]; it is non-trivial to make accurate prediction and add new VMs into the system ahead of the long setup time. In fact our work complements both the reactive and predictive approaches. No matter which approach is chosen, when a controller decides to scale out/in, reallocating soft resources is necessary to maximize the efficiency of underlying hardware resources.

Queuing Model for Performance Prediction. Modeling of n-tier applications has been studied for performance prediction and system management. Urganonkar et al.[37] propose a queue-based model to capture the performance characteristics of each tier and application idiosyncrasies. This model focuses on the relationship between number of sessions and average response time, while our work focuses on the concurrency management in each tier. Newell et al.[38] present a latency oriented model, which is applied in the SEDA-based single server environment. Their main focus is to optimize the threads allocation among different stages in one server while our problem domain is in n-tier application scaling management in cloud. Franks et al. [39], [40] propose a layered queuing network model which characterizes the dependencies of software and hardware resources across nodes in different tiers of a distributed system. However, their model does not take the impact of workload concurrency on the sensitivity of server performance into account. Our research focus is to ensure the optimal request processing concurrency in the system to efficiently utilize the critical hardware resource even after system scaling.

Software Performance Engineering. Software Engineering approaches has been explored for system performance optimization. For example, Zheng et al. [41] proposed an auto-generation technique of configuration files for Internet services such as n-tier applications. Their focus is to remove various mis-configuration caused by manual operations. On the other hand, our objective is to optimize performance through soft resources on-line adaptation in the system scaling management. Gunther et al.[17] proposed a methodology to characterize the relationship between server performance and threads concurrency in a single server environment. Maji et al.[42] investigated some important parameters (e.g., *MaxClients* and *KeepaliveTimeout*) of an Apache web server and see how they affect the server's performance in a shared cloud environment. Their focus is to reduce the interference from the co-hosted VMs by reconfiguring those parameters.

7 Conclusion

In this paper we show the importance of soft resource allocations in scaling n-tier applications in the cloud. Through extensive experiments using a representative n-tier web application benchmark (RUBBoS), we demonstrate that scaling an n-tier system by adding or removing VMs without appropriately re-allocating soft resources (e.g., server threads and connections) may lead to significant performance degradation of the system (Section 2.2). We build a concurrency-aware model that combines some operational queuing laws and the fine-grained measurement data of the system to determine a near optimal concurrency setting of each tier in the system (Section 3). We integrate the model into our dynamic concurrency management (DCM) framework to intelligently reallocate soft resources of each tier during the system scaling process (Section 4). Our experiments using six real-world bursty workload traces demonstrate that DCM can achieve significantly shorter tail latency while higher resource efficiency compared to hardware-only scaling solutions (Section 5).

References

[1] "memcached - a distributed memory object caching system," "https://memcached.org//".

[2] 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)*, vol. 30, no. 4, p. 14, 2012.

[3] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes, "Agile: Elastic distributed resource scaling for infrastructure-as-a-service," in *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, 2013, pp. 69–82.

[4] C. Z. Xu, J. Rao, and X. Bu, "URL: A unified reinforcement learning approach for autonomic cloud management," *Journal of Parallel and Distributed Computing*, vol. 72, no. 2, pp. 95–105, 2012.

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

[6] E. Barrett, E. Howley, and J. Duggan, "Applying reinforcement learning towards automating resource allocation and application scalability in the cloud," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 12, pp. 1656–1674, 2013.

[7] 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-09-12*, 2012.

[8] Q. Wang, S. Malkowski, D. Jayasinghe, P. Xiong, C. Pu, Y. Kanemasa, M. Kawaba, and L. Harada, "The impact of soft resource allocation on n-tier application scalability," *Proceedings - 25th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2011*, pp. 1034–1045, 2011.

[9] OW2, "Rubbos," http://forge.ow2.org/projects/rubbos/.

[10] S. Adler, "The slashdot effect: An analysis of three internet publications," http://ldp.dvo.ru/LDP/LG/issue38/adler1.html, Mar. 1999.

[11] Apache, "Jmeter," http://jmeter.apache.org/.

[12] P. J. Denning and J. P. Buzen, "The operational analysis of queueing network models," *ACM Comput. Surv.*, vol. 10, no. 3, 1978.

[13] J. Dilley, R. Friedrich, T. Jin, and J. Rolia, "Measurement tools and modeling techniques for evaluating web server performance," in *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Springer, 1997, pp. 155–168.

[14] J. Bi, Z. Zhu, R. Tian, and Q. Wang, "Dynamic provisioning modeling for virtualized multi-tier applications in cloud data center," in *2010 IEEE 3rd International Conference on Cloud Computingofol*. IEEE, 2010, pp. 370–377.

[15] 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.

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

[17] N. J. Gunther, S. Subramanyam, and S. Parvu, "A methodology for optimizing multithreaded system scalability on multi-cores," *CoRR*, vol. abs/1105.4301, 2011.

[18] 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 *Proceedings of the 25th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2011)*, 2011, pp. 1034–1045.

[19] J. Kreps, N. Narkhede, J. Rao *et al.*, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, 2011, pp. 1–7.

[20] Q. Wang, Y. Kanemasa, J. Li, D. Jayasinghe, T. Shimizu, M. Matsubara, M. Kawaba, and C. Pu, "Detecting transient bottlenecks in n-tier applications through fine-grained analysis," in *Proceedings of the 6th International Conference on Cloud computing (Cloud 2013)*, 2013.

[21] S. Malkowski, M. Hedwig, and C. Pu, "Experimental evaluation of n-tier systems: Observation and analysis of multi-bottlenecks," in *IISWC '09*.

[22] "Haproxy," http://www.haproxy.org/.

[23] Amazon, "Ec2 autoscaling," https://aws.amazon.com/autoscaling/.

[24] A. H. Mahmud and S. Ren, "Online capacity provisioning for carbon-neutral data center with demand-responsive electricity prices," *SIGMETRICS Perform. Eval. Rev.*, vol. 41, no. 2, pp. 26–37, Aug. 2013.

[25] S. Ren and Y. He, "Coca: Online distributed resource management for cost minimization and carbon neutrality in data centers," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 39:1–39:12.

[26] L. Zhang, Y. Zhang, P. Jamshidi, L. Xu, and C. Pahl, "Workload patterns for quality-driven dynamic cloud service configuration and auto-scaling," in *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, ser. UCC '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 156–165.

[27] "Amazon auto scaling group limits," https://docs.aws.amazon.com/autoscaling/ec2/userguide/as-account-limits.html.

[28] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal, "Dynamic provisioning of multi-tier internet applications," in *Second International Conference on Autonomic Computing (ICAC'05)*, 2005, pp. 217–228.

[29] ITA., "The internet traffic archives:wordcup98," http://ita.ee.lbl.gov/html/contrib/WorldCup.html, 1998.

[30] NLANR, "National laboratory for applied network research. anonymized access logs," ftp://ftp.ircache.net/Traces/., 1995.

[31] N. Mi, G. Casale, L. Cherkasova, and E. Smirni, "Injecting realistic burstiness to a traditional client-server benchmark," in *Proceedings of the 6th international conference on Autonomic computing*, 2009.

[32] A. Gandhi, T. Zhu, M. Harchol-Balter, and M. A. Kozuch, "Softscale: Stealing opportunistically for transient scaling," in *Proceedings of the 13th International Middleware Conference*, ser. Middleware '12. New York, NY, USA: Springer-Verlag New York, Inc., 2012, pp. 142–163.

[33] H. Jayathilaka, C. Krintz, and R. Wolski, "Response time service level agreements for cloud-hosted web applications," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 2015, pp. 315–328.

[34] D. Serrano, S. Bouchenak, Y. Kouki, T. Ledoux, J. Lejeune, J. Sopena, L. Arantes, and P. Sens, "Towards qos-oriented sla guarantees for online cloud services," in *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*. IEEE, 2013, pp. 50–57.

[35] E. B. Lakew, E. Elmroth *et al.*, "Service level and performance aware dynamic resource allocation in overbooked data centers," in *Cluster, Cloud and Grid Computing (CCGrid), 2016 16th IEEE/ACM International Symposium on*. IEEE, 2016, pp. 42–51.

[36] L. Wang, J. Xu, H. A. Duran-Limon, and M. Zhao, "Qos-driven cloud resource management through fuzzy model predictive control," in *Autonomic Computing (ICAC), 2015 IEEE International Conference on*. IEEE, 2015, pp. 81–90.

[37] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi, "An analytical model for multi-tier internet services and its applications," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 33, no. 1. ACM, 2005, pp. 291–302.

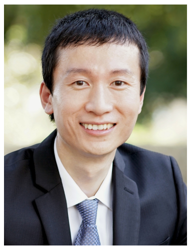
[38] A. Newell, G. Kliot, I. Menache, A. Gopalan, S. Akiyama, and M. Silberstein, "Optimizing distributed actor systems for dynamic interactive services," in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16. ACM, 2016, pp. 38:1–38:15.

[39] G. Franks, D. Petriu, M. Woodside, J. Xu, and P. Tregunno, "Layered bottlenecks and their mitigation," in *Quantitative Evaluation of Systems, 2006. QEST 2006. Third International Conference on*. IEEE, 2006, pp. 103–114.

[40] G. Franks, T. Al-Omari, M. Woodside, O. Das, and S. Derisavi, "Enhanced modeling and solution of layered queueing networks," *IEEE Transactions on Software Engineering*, vol. 35, no. 2, pp. 148–161, 2009.

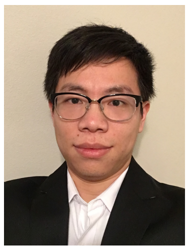
[41] W. Zheng, R. Bianchini, and T. D. Nguyen, "Automatic configuration of internet services," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, p. 219, 2007.

[42] A. K. Maji, S. Mitra, B. Zhou, S. Bagchi, and A. Verma, "Mitigating interference in cloud services by middleware reconfiguration," *Proceedings of the 15th International Middleware Conference on - Middleware '14*, pp. 277–288, 2014.

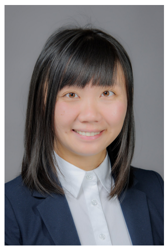


Qingyang Wang is an Assistant Professor in the Department of EECS at Louisiana State University-Baton Rouge. His research is in distributed systems and cloud computing with a current focus on performance and scalability analysis of large-scale web applications (e.g., Amazon.com). He has led research projects at LSU on cloud performance measurements, scalable web application design, and automated system management in clouds. He graduated from the College of Computing, Georgia Institute of Technology with a Ph.D. degree in 2014, and has previously received his MSc and BSc degrees in computer science and engineering from Chinese Academy of Sciences and Wuhan University in 2007 and 2004, respectively. He is a recipient of the Best Student Paper award in IEEE Cloud 2011.

Hui Chen Hui Chen received the Bachelors and the PhD degrees from Beijing University of Posts and Telecommunications, Beijing, China, in 2006 and 2012, respectively. He is currently a research staff in the 2012 Lab of Huawei Company. Before joining Huawei, he has worked as an assistant researcher in Louisiana State University, Auburn University and Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences for totally five years. His research interests include cloud computing, energy efficiency management of data center and big data processing.



Shungeng Zhang is a Ph.D. student in the Department of EECS at Louisiana State University-Baton Rouge. Currently, he is presently working in the cloud computing lab as a research assistant under Dr. Qingyang Wang. His research interest lies in performance and scalability analysis of asynchronous Internet server architecture, with the aim of achieving highly responsive web applications running at high utilization in cloud. He graduated from the School of Software Engineering, HuaZhong University of Science & Technoogy with a B.Eng. degree in 2014.



Liting Hu got her PhD degree in Computer Science at Georgia Institute of Technology. Before that, she completed her undergraduate degree in Computer Science at Huazhong University of Science and Technology in China. Her research is in the general area of distributed systems and its intersection with big data analytics, resource management, power management and system virtualization. She spent summers interning at IBM T.J. Watson Research Center, Intel Science and Technology Center for Cloud Computing, Microsoft Research Asia, VMware, and has been working closely with them.

Balaji Palanisamy Balaji Palanisamy received the MS and PhD degrees in computer science from the College of Computing, Georgia Tech, in 2009 and 2013, respectively. He is an assistant professor at the School of Information Science, University of Pittsburgh. His primary research interests lie in scalable and privacy-conscious resource management for large-scale distributed and mobile systems. At University of Pittsburgh, he codirects research in the Laboratory of Research and Education on Security Assured Information Systems (LERSAIS). He received the Best Paper Award at the Fifth International Conference on Cloud Computing, 2012. He is a member of the IEEE and is currently the chair of the IEEE Communications Society Pittsburgh Chapter.