DCM: Dynamic Concurrency Management for Scaling n-Tier Applications in Cloud

Hui Chen†, Qingyang Wang†, Balaji Palanisamy*, Pengcheng Xiong‡

†*Computer Science and Engineering, Louisiana State University*

**School of Information Science, University of Pittsburgh*

‡*Hortonworks*

***Abstract*—Scaling web applications such as e-commerce in cloud by adding or removing servers in the system is an important practice to handle workload variations, with the goal of achieving both high quality of service (QoS) and high resource efficiency. Through extensive scaling experiments of an n-tier application benchmark (RUBBoS), we have observed that scaling only hardware resources without appropriate adaptation of soft resource allocations (e.g., thread or connection pool size) of each server would cause significant performance degradation of the overall system by either under- or over-utilizing the bottleneck resource in the system. We develop a dynamic concurrency management (DCM) framework which integrates soft resource allocations into the system scaling management. DCM introduces a model which determines a near-optimal concurrency setting to each tier of the system based on a combination of operational queuing laws and online analysis of fine-grained measurement data. We implement DCM as a two-level actuator which scales both hardware and soft resources in an n-tier system on the fly without interrupting the runtime system performance. Our experimental results demonstrate that DCM can achieve significantly more stable performance and higher resource efficiency compared to the state-of-the-art hardware-only scaling solutions (e.g., Amazon EC2-AutoScale) under realistic bursty workload traces.**

I. INTRODUCTION

Scalability has become one major advantage of cloud data centers where application owners can dynamically scale the underlying computing resources on the fly if necessary. First, many web applications are provided using the n-tier architecture, in which each tier of the system can be easily extended by adding or removing servers. Second, web applications naturally have bursty workload, where the peak workload in rush hours can be 10X higher than the overall average [1], [2]; over-provisioning only for peak workload can waste significant amount of computing resources and power. Therefore, scalability is extremely important for web applications to achieve high resource efficiency.

Achieving high resource efficiency through cloud scalability is a significant challenge because web applications usually have strict service level agreements (SLAs) such as bounded response time. Given the burstiness nature of the workload for web applications, intelligent matching of computing resources with the dynamically changing workload to always meet the SLAs is a non-trivial task. Various dynamic resource scheduling methods [3], [4], [5], [6], [7] have been proposed to scale the system by adding/removing servers along with the real-time workload variation. These previous research efforts mainly focus on when (e.g., through workload prediction) and how (e.g., through virtual machine live migration) to add/remove hardware resources, without seriously considering software reconfiguration after the system scaling. On the other hand, previous research [8] shows that soft resources such as server threads and database connections that control the request processing concurrency in the system have a significant impact on n-tier application performance.

Consequently, we develop a dynamic concurrency management (DCM) framework which integrates soft resource allocations into the system scaling management. DCM introduces a concurrency-aware model which is able to determine a near-optimal concurrency setting of the servers in each tier based on a combination of operational queuing laws and online analysis of fine-grained measurement data (e.g., throughput and request processing concurrency) of each server in the system. We implement DCM as a two-level actuator which scales both the hardware and the soft resources in an n-tier web system on the fly without interrupting the runtime system performance. The first-level is to start new VMs or remove the idle ones as the traditional hardware-only scaling strategies do. The second-level is to support runtime adjustment of soft resource allocations of the servers under control, based on the prediction of the derived concurrency-aware model.

The rest of this paper is organized as follows. Section II illustrates the impact of thread allocations on the performance of Tomcat and MySQL. Section III introduces our concurrency-aware model which predicts the optimal concurrency setting in each tier of the system. Section IV presents the design and implementation of the DCM framework. Section V shows the evaluation results. Section VI discusses related work and Section VII concludes the paper.

II. BACKGROUND AND MOTIVATION

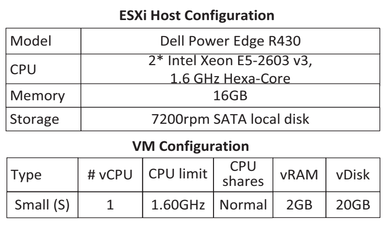
A. *Experiment Environment*

We use the RUBBoS benchmark [9], a standard n-tier benchmark simulating the popular news website Slashdot [10]. The RUBBoS benchmark application can be deployed as three-tier illustrated in Fig. 1(c) or four-tier (with the load balancer tier for databases). There are 24 servlets which provide different interaction services. All the servlets can be divided into two categories: CPU intensive browse-only or disk I/O intensive workload. We use the CPU intensive browse-only workload in this paper.

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



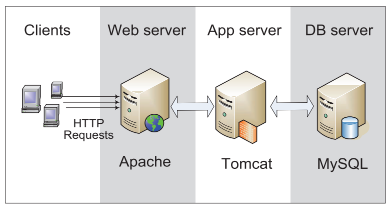
ESXi Host Configuration

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 Profile



(c) 1/1/1 Sample Topology

Fig. 1: Experimental setup information

All the experiments are conducted in our private VMWare ESXi cluster. Fig. 1 shows the experimental setup in our experiments. We use a three-digit notation #W/#A/#D to denote the number of Apache web, Tomcat application, and MySQL servers launched in the experiment. For each hardware configuration #W/#A/#D, we use $\#W_T/\#A_T/\#A_C$ to denote three representative soft resources in the system: the thread pool size in an Apache server, thread pool and database (DB) connection pool size in a Tomcat server. These three types of soft resources control the maximum concurrency of request processing in Apache, Tomcat, and MySQL, respectively. We note that the original RUBBoS implementation uses one DB connection pool per servlet; we modified it to let all the servlets share one global DB connection pool in order to precisely control the number of concurrent requests flowing to the downstream MySQL. We also developed agent tools to enable runtime scaling of the aforementioned soft resources.

We use three types of workload generators: Jmeter [11], the original RUBBoS workload generator which simulates realistic workload under a static number of concurrent users, and the revised RUBBoS client emulator which simulates realistic workload under a dynamically changing number of concurrent users based on a workload trace file. The original RUBBoS workload generator is used to test the system performance under realistic static workload environment. In contrast, the revised RUBBoS workload generator is to test the system stability under realistic dynamic and bursty workload.

B. Performance Degradation with Sub-Optimal Concurrency Setting

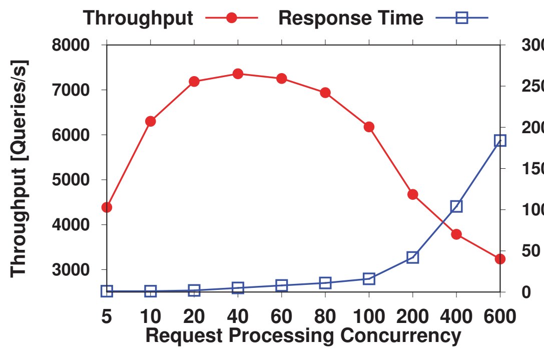
An n-tier application such as e-commerce (e.g. Amazon.com) commonly process hundreds or even thousands of HTTP requests from clients concurrently, and the concurrency level of request processing in the system is mainly restricted by soft resource allocations such as a server thread pool or a database connection pool in each component server. In this section we show concrete experimental results to illustrate the impact of soft resource allocations on the performance of typical MySQL server in an n-tier application as in Fig. 2(a). In this experiment, we use Jmeter to generate workload (requests extracted from the RUBBoS workload trace) with precisely controlled concurrency to stress the MySQL (Fig. 2(a)) server. For each level of workload concurrency, we allocate the matching thread pool size in the corresponding server so that the workload concurrency matches the request processing concurrency in the server. Fig. 2(a) shows the case of the MySQL server as we increase the request processing concurrency from 5 to 600. The interesting observation is that the MySQL throughput reaches the peak as the request processing concurrency increases to 40, and then decreases significantly as the request processing concurrency continues to increase.

The sensitivity of MySQL server performance on request processing concurrency also explains the unexpected performance degradation of the 3-tier system after scaling out as shown in Fig. 2(b). In this set of experiments, the system hardware configuration is 1/1/1 at the beginning, with one Apache web server, one Tomcat server, and one MySQL server. The soft resource configuration is the default 1000/100/80, denoting 1000 threads in Apache, 100 threads and 80 database connections in Tomcat. Under this configuration, the maximum concurrency level of request processing in MySQL is limited by the Tomcat DB connection pool size (80). Since Tomcat is the bottleneck tier in the 1/1/1 configuration, we scale the system to 1/2/1 by adding one more Tomcat with the default soft resource allocation (i.e., 80 DB connection pool size) to handle the increased workload, however, the maximum number of concurrent requests flowing to MySQL unexpectedly doubles (160) due to the newly added Tomcat server, leading to the poor performance in MySQL as illustrated in Fig. 2(a). To efficiently utilize the new added Tomcat server and improve the overall system performance after scaling, we need to reallocate the database connection pool size in each Tomcat server to 20 as that the maximum concurrency in MySQL is limited to 40, with which MySQL is able to achieve better throughput.

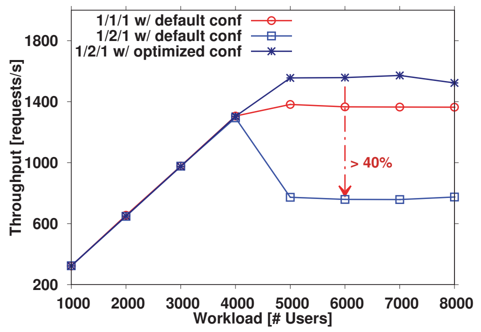
The above experimental results illustrate the importance of soft resource allocations on the n-tier application scaling management in cloud; only scaling hardware resources in an n-tier system without appropriate soft resource adaptation can lead to significant performance degradation, even though more hardware resources are added into the system.

III. Concurrency-aware Model for n-Tier Application Performance

In this section we explain a concurrency-aware model which is able to predict the optimized soft resource allocation of



(a) MySQL achieves reasonable performance when the request processing concurrency is between 20 to 80.



(b) The system throughput significantly decreased under high workload after scaling-out if no software configuration changes are adopted.

Fig. 2: Impact of request processing concurrency on the performance of typical servers in an n-tier system. (a) shows that both too low and too high request processing concurrency in MySQL can lead to poor performance, (b) illustrate the influence of scaling out Tomcat tier inappropriately may lead to worse performance .

each component server in the system under different hardware configurations. The model has two enhancements compared to the classic queuing model: first, it captures the basic request flow processing characteristics in an n-tier application, for example, an HTTP request may trigger multiple interactions between component servers in the system; second, it considers the non-trivial impact of multi-threading overhead on n-tier application performance as we have shown in Section II-B. Our goal is to maximize the whole system throughput through optimized soft resource allocations in each tier, even if the system scales to a different size.

A. Concurrency-Aware Queue Model

Consider that an application with M tiers is denoted by T_1,...,T_M. The number of servers in tier T_m is denoted as K_m, where $1 \leq m \leq M$. For simplicity, we assume each tier only has one server at the beginning, so the server resource unit of each tier is 1. Let U_m represent the server utilization in tier T_m. According to the Utilization Law and Forced Flow Law, we could get the following equation for each tier:

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

In Equation 1, S_m represents the average request service time at tier m, X means the throughput of whole system, X_m is the throughput of T_m, V_m indicates the visit ratio from upstream tier. Based on above equations, we could get the system throughput:

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

The visit ratio V_m depends on the workload characteristics. For example, a sample HTTP request to the Apache web server triggers one AJP request to Tomcat, but two subsequent queries to the backend MySQL. In this case, we get $V_2 = 1$ and $V_3 = 2$. S_m means the average service demand of each sub request for tier T_m, thus $V_m * S_m$ means the overall service demand of an HTTP request for T_m. Assume there is only one server in each tier, we could easily figure out that the bottleneck tier of the whole system is the tier that has the longest service demand $\max_{1 \leq m \leq n} (V_m * S_m)$. Suppose T_b is the bottleneck tier, then the maximum system throughput is achieved when $U_b = 1$, meaning the bottleneck tier resource is 100% utilized. The maximum system throughput X_{max} can be expressed as follows:

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

In reality, there is K_b servers in the bottleneck tier, thus the above equation can be transformed to

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

where γ is the correction parameter to the linear increase of servers in the bottleneck tier. As we know the system performance will not double if we increase the bottleneck tier resource from one server to two due to many other factors, such as the load inbalancing problem among servers or the sharing usage of downstream resources.

Given a fixed system configuration and stable workload characteristics, K_b and V_b can be determined. So according to Equation 4, X_{max} can be predicted once S_b is determined. However, it is non-trivial to trace the real service time of an HTTP request in each tier, which may involve multiple interactions with other components in the system. Especially when we consider about the multi-threading situation, resource contention may happen when multiple threads operate on shared resources, which may change the original service time in a single-threaded environment as it has been discussed in many previous work [12], [13], [14]. So we use another method to estimate the average service time of HTTP requests in each tier.

B. Service Time Model in Multi-Threading Environment

Assume S_b^0 is the service time of a single-threaded server in the bottleneck tier. When we consider about the impact incurred by multiple threading to the service time S_b^0, there are two sources of latency deserving our attention. One is threads contention, the other is crosstalk penalty, which is also called consistency or coherency penalty.

Threads contention is determined by the multi-threading implementation in the CPU architecture design. The most common choice is called Simultaneous Multi-Threading (SMT) [15], which is a variation of fine-grained multi-threading. The characteristics of fine-grained multi-threading is to switch between threads on each clock, causing the execution of instructions from multiple threads to be interleaved, which is often done in a round-robin fashion. Based on this observation, the delay caused by threads contention can be modeled as linear increasing with the number of threads.

Crosstalk penalty is due to the consistency/coherency requirements in a multi-threading environment, which has been discussed in Hennessy and Patterson's work [15]. Whether it is distributed or directory-based shared memory coherence model, the worst case involves N*(N-1) invalid messages if each thread wants to write a new value to a shared variable. The penalty grows exponentially as the number of threads increases. By combining the above two sources of delay, we get the adjusted service time with N_b threads as follows:

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

The parameters α_b, β_b are related to various factors such as workload characteristics and hardware specification. We note that when $N_b = 1$, Equation 5 reverses back to the single-threaded scenario where $S_b^* = S_b^0$.

Although multi-threading leads to longer delay for the execution of each individual request due to threads contention and crosstalk penalty, it takes full advantage of the pipelined components in modern CPU architecture, which could help fully utilize the CPU resource and increase system throughput. As the notebook [15] illustrates, the N_b threads take turns to use CPU cycles, and it takes S_b^* time to finish each request. But the total throughput during this period is N_b. Then the adjusted average service time in a multi-threading environment is:

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

Based on 4 and 6, we could get the relationship between the system's maximum throughput and the concurrency in the bottleneck tier as follows:

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

C. Maximizing System Throughput From the Model

According to Equation 4, to maximize the whole system throughput X_{max}, we need to minimize S_b. So we transform Equation 6 to the following:

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

The minimum S_b is achieved when when $N_b = \sqrt{\frac{S_b^0 - \alpha_b}{\beta_b}}$. Then we turn Min($S_b$) back to Equation 4, and we get:

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

In this model, we can find the optimal operation point for the whole system is to set the threads of each server in the bottleneck tier as N_b, which depends on the parameters S_b^0, α_b, and β_b. We can determine these parameters via online monitoring of the whole system, then regress based on the measured system throughput and the thread allocation of each server in the bottleneck tier. We note that N_b is the theoretical optimal concurrency setting, the realistic configuration of "maxThreads" or "maxConnections" should be larger than this theoretical value because not all threads will be in Active state during the operation.

IV. Dynamic Concurrency Management Design and Implementation

The DCM architecture is illustrated in Fig. 3. It contains three components: Fine-Grained Resource Monitor, Optimization Controller, and Actuators.

- **Fine-Grained Resource Monitor:** We install a monitoring agent in each VM to collect both the system-level metrics (e.g., CPU, disk I/O, Memory) and the application-level metrics (e.g., average server throughput, response time, active threads number (concurrency)). Each monitoring agent continuously sends the collected data back to a storage server (Kafka [16]) at every one second. Then the controller will consume these data as needed. Because the data producer and consumer may operate in different rates, we use Kafka as an intermediate storage server to coordinate the data production and consumption between the monitor agents and the controller.
- **Optimization Controller:** The controller analyzes the data from Kafka and makes adaptation decisions based on the concurrency-aware model as we present in Section III. Usually the controller needs to make two decisions at the moment of burst workload. The first decision is the VM-level scaling: when to launch new VMs to support the bottleneck tier, or turn off idle VMs to save cost. The second decision is how to adapt soft resource allocations after the VM-level scaling is done. In our current implementation, we adopt the resource-usage driven controller in the VM-level scaling, which means Actuators will be called when the resource usage of some tier exceeds a predefined threshold (e.g., 80%).
- **Actuators:** There are two actuators in this component. One is VM-agent, which is responsible for starting new VMs or removing the idle ones. The other is APP-agent, which is used to do fine-grained adaptation of soft resource allocations. Usually the VM-agent will be called first, followed by the APP-agent.

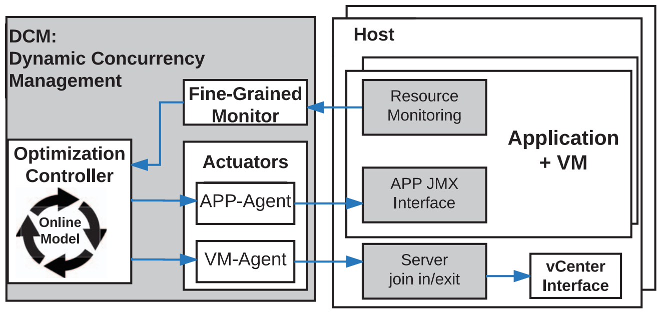


Fig. 3: The DCM Architecture

A. VM-Agent for VM-level Scaling

VM-agent is for the VM-level scaling. In a cloud computing environment, starting or turning off VMs is easy by just remotely calling the corresponding APIs of the underlying hypervisor. One factor that makes the scaling complicated is the component servers that run inside VMs. For example, adding VMs that run stateless components such as web servers or application servers is relatively easy because they can join the existing system and start processing new requests seamlessly. However, adding VMs that run stateful servers is more complicated because of the data/state consistency issues. In our implementation, we set the preparation period as 15 seconds for a VM to be in the service mode after VM-agent decides to start the VM.

Another challenging is how to rebalance the load to the tiers after scaling. There usually is a load balancer in front of the tier that are able to scale. In our experiments we adopt HAproxy [17] as a load balancer for the application server tier and the database server tier.

B. APP-Agent for Soft Resource Re-Allocation

APP-agent is to control the concurrency of request processing in each component server through readjusting the soft resource allocations in the system after the VM-level scaling is done. There are two ways to change a server's concurrency of request processing, one is adjusting the server's thread pool (STP) size and the other is controlling the connection pool size of the corresponding upstream tier. In our implementation we adopt the first way (i.e., adjusting the STP size) to control the concurrency of request processing in Tomcat because the application server tier may serve HTTP requests from clients directly (or through HAproxy), the concurrency of which is out of our control. On the other hand, we adopt the second way to control the concurrency of request processing in MySQL since we are able to adjust the DB connection pool (DBConnP) size in Tomcat, which is the upstream tier of MySQL.

V. Experimental Evaluation

In this section we run experiments to train and validate the concurrency-ware model for the optimal soft resource allocation for both Tomcat and MySQL. Then we apply the model into our DCM framework (see Section IV) and test the effectiveness of DCM in system scaling management under the "large variation" workload scenario. Concretely, we compare the performance of a 3-tier web application under two system scaling management strategies: one is with DCM and the other is with the traditional hardware-only scaling strategy (e.g., "EC2-AutoScale") which has been widely used by academic researchers [3], [18] and industry practitioners [19]. We will see that the system with DCM is able to achieve much more stable performance than that with the hardware-only scaling strategy.

TABLE I: model training parameters and prediction result

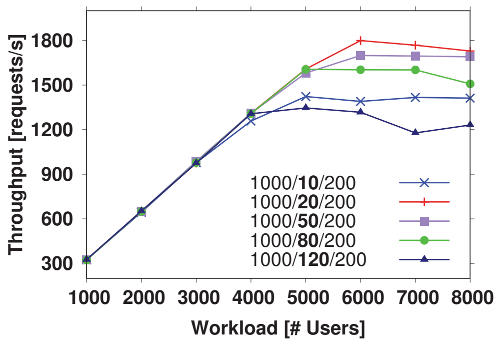
	Tomcat Model	MySQL Model
S_b^0	2.84e-02	7.19e-03
α_b	9.87e-03	5.04e-03
β_b	4.54e-05	1.65e-06
γ	11.03	4.45
R^2	0.96	0.97
N_b	**20**	**36**
X_{max}	946	865

A. Model Training and Validation

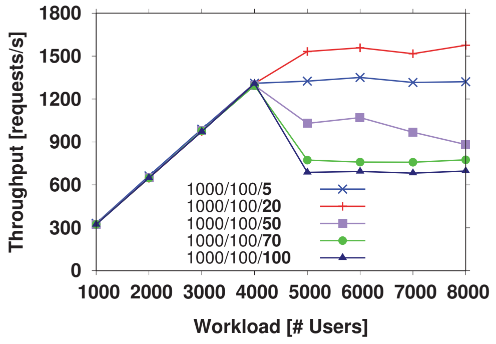
In the training phase of experiments, we use Jmeter to generate workload with different concurrencies to train the model. Because we set the think time between consecutive HTTP requests from the same thread to be zero, the workload concurrency for the target system can be controlled by the number of concurrent users specified in Jmeter. Then in the validation phase of experiments, we change back to the original RUBBoS workload, which is more close to realistic production workloads. We validate whether the optimal soft resource allocation predicted by the model is able to achieve the maximum system throughput.

Model training for Tomcat: The purpose of the model is to characterize the relationship between the system throughput and the concurrency of request processing in Tomcat. To achieve this goal, we need to design an experiment which makes Tomcat the bottleneck tier of the system, so we choose the 1/1/1 hardware configuration. The soft resource allocation is the default 1000-100-80. We use Jmeter to generate workload with concurrency from 1 to 200. The <*concurrency, throughput*> pairs are the input to train the model as shown in Equation 7. We use the Least-Square Fitting method to estimate the parameters in Equation 7 and we get the values of the parameters as presented in Table I. The "knee" point for the maximum throughput is achieved when the request processing concurrency $N_b = 20$. Also the statistical R-Squared value is 0.96 based on our extra measurement data, which indicates that the model has a high accuracy of predicting system throughput under different Tomcat concurrencies.

In order to validate the generality of the model, we have conducted experiments with the original RUBBoS clients which simulate the realistic workload scenario with averagely 3-second "think" time between consecutive requests from the same user. Fig. 4(a) shows the experimental results. We have chosen five representative soft resource allocations which



(a) Throughput comparison with varied number of Tomcat threads in the 1/1/1 case



(b) Throughput comparison with varied number of Tomcat DB connections in the 1/2/1 case

Fig. 4: Model validation under the 1/1/1 and the 1/2/1 configuration. (a) and (b) show that the optimal soft resource allocation (20 for Tomcat and 36 for MySQL) predicted by our concurrency-aware model indeed outperforms other allocations, including the default one. We note that the 1000/100/18 case achieves the best in the 1/2/1 configuration because there are two Tomcat servers.

includes the optimal solution 1000/20/80 predicted by the model. We can see that the system throughput in the "optimal" configuration case indeed outperforms the other cases. Specially, the throughput of the "optimal" configuration case achieves 30% more throughput than the default configuration case (100 Tomcat threads).

Model training for MySQL: We choose the 1/2/1 hardware configuration to train the model for MySQL because MySQL is the bottleneck tier of the system under this configuration. We repeat the same process of model training for MySQL as we have done for Tomcat. Table I also shows the estimated values of the parameters of the MySQL model with the R-Squared value 0.97. The maximum system throughput is achieved when $N_m = 36$. We also validate the generality of the model using the original RUBBoS clients which simulate the realistic workload scenario. Fig. 4(b) shows that the throughput of the system with the optimal configuration (1000-100-18) suggested by the model (each Tomcat share half of the optimal connection pool size) indeed outperforms the other four representative cases, including the default configuration case (80 database connections).

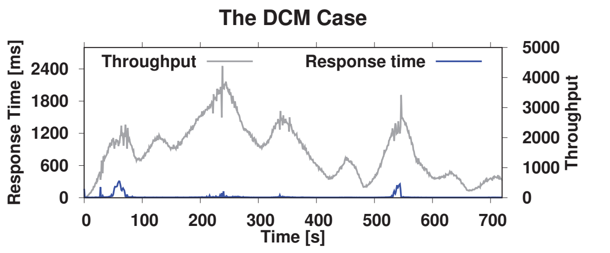
B. Evaluation of DCM with Concurrency-Aware Models

In this section we show the effectiveness of our DCM framework in stabilizing the system performance during system scaling in the face of realistic bursty workloads. We have implemented two system scaling controllers. One is with DCM and the other is called EC2-AutoScale [19] which follows the EC2 Autoscaling strategy provided by Amazon AWS service. EC2-AutoScale enables customers to set the threshold value (usually the CPU utilization) to dynamically add or remove VMs to an Auto Scaling group. The control policy relies on Amazon CloudWatch to send alarms and trigger scaling activities. For both controllers we set the control period to be 15-second, which is similar as in other state-of-the-art control policies [20]. We also adopt the "quick start but slow turn off" scaling policy learned from the AutoScale work [3] to avoid performance instability under bursty workload. Concretely, if the concerned resource utilization of a tier is higher than an upper bound threshold (e.g., 80%) during one control period, the controller triggers the event of turning on a VM, which will spend another 15-second preparation period before joining the system and start servicing requests. On the other hand, we turn off one VM in a tier only if the utilization of all the concerned resources in the tier is lower than a lower bound threshold (e.g., 40%) in three consecutive control periods.

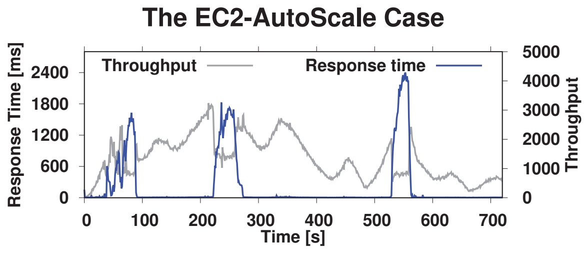
In this experiment, we use the revised RUBBoS client emulator as we present in Section II-A to genearte realistic bursty workload. The burstiness of workload is controllable and reproducible in our experiments based on the workload trace file. The "Large Variation" workload trace is chosen from the AutoScale paper [3], which is categorized and published by Gandhi et al.. We start the system with 1/1/1 configuration and soft resource allocation with 1000-200-40, which means database connection pool are configured as the optimal value at the beginning. Then we get the performance results of "DCM" and "EC2-AutoScale" under the burstiness workload as presented in Fig. 5.

Taking the first period 50s$\sim$90s as an example, we found a large response time spike (over 1 second) and throughput drop in the EC2-AutoScale case as shown in Fig. 5(b). Meanwhile, the Tomcat tier scales from one server to two at 67s due to the increased workload and the CPU utilization exceeds the trigger line (see Fig. 5(d)). [1] However, the system performance get even worse after the second Tomcat instance adds in. The reason can be found in Fig. 5(f). Once the second Tomcat is added to the system, the bottleneck tier shifts to MySQL. At the same time, the maximum concurrent requests flowing to MySQL also increase from the original 80 (the default DB connection pool size in Tomcat) to 160 due to the new added Tomcat. High concurrency in MySQL leads to low efficiency of MySQL, and eventually the overall system performance degradation. The second performance

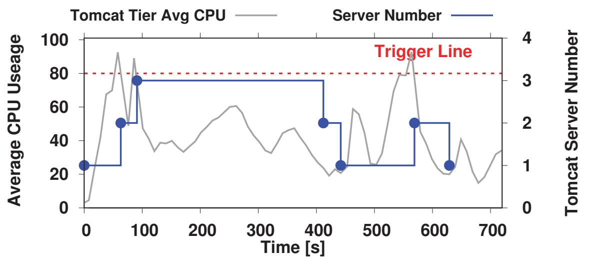
[1]The system performance deteriorates before the scaling activity is because of the 15-second control period, causing the delay of the scale-out activity.



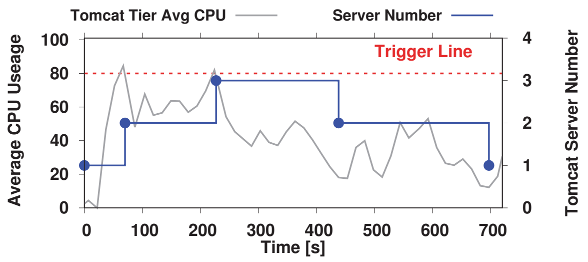
(a) Stable system response time under the "Large Variation" bursty workload. The system throughput also follows the workload trend, indicating no throughput loss.



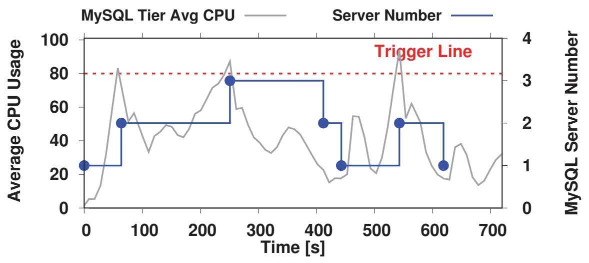
(b) Large response time variation under the same bursty workload trace as in (a). There is also throughput drop during each of the three response time spikes at 70s, 250s, and 550s.



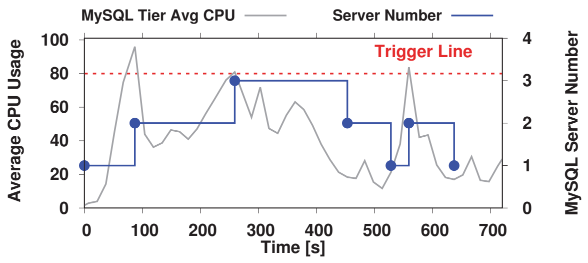
(c) The Tomcat tier scales with the variation of CPU util. The scaling-out activities of Tomcat at 80s and 100s do not interrupt system performance significantly as shown in (a).



(d) In the EC2-AutoScale case, the scaling-out activities of Tomcat at 80s and 220s coincide with the large response time spikes and throughput drops as shown in (b).



(e) The MySQL tier scales with the variation of CPU util. The scaling-out activities of MySQL at 80s, 220s, and 540s do not interrupt system performance significantly as shown in (a).



(f) In the EC2-AutoScale case, the scaling-out activities of MySQL at 80s, 280s, and 550s coincide with the large response time spikes and throughput drops as shown in (b).

Fig. 5: Performance comparison between DCM and EC2-AutoScale under the same "Large Variation" workload. (a)(c)(e) are for DCM and (b)(d)(f) are for EC2-AutoScale. The system starts at the same initial configuration (1/1/1 with the default soft resource allocations), however, the DCM case achieves much more stable performance than the EC2-AutoScale case by comparing (a) and (b).

deterioration period 227s∼259s has the same reason as the first one when the third Tomcat and third MySQL are added to the system subsequently to handle the increased workload. The third period 530s∼560s is different from the previous two. The MySQL tier changes from two instances to one at 528s because of continuous low workload. Suddenly high volume of workload floods into MySQL during the next control period, so the only MySQL instance encounters high concurrency (160), causing low QPS of MySQL.

In contrast, the DCM case only has moderate performance variation during the three periods mentioned above. This is because DCM dynamically adjusts the concurrency of both Tomcat and MySQL to the "optimal" level, thus both of them perform at the maximum speed during the temporarily overloaded periods, leading to more stable performance of the system when compared to the EC2-AutoScale case.

VI. Related Work

Most previous research efforts on dynamic capacity management in cloud environments [3], [21] share the same goal: providing a satisfactory quality of service (QoS) while minimizing the cost, which could be energy consumption, financial cost, or both. Their approaches can be divided into two categories: feedback control based reactive approaches and predictive approaches. Reactive approaches [3] use different feedback signals such as the current system response time, CPU utilization, or processing queue length to decide when to scale out/in the system. In the scale out scenario, the performance damage already occurs before the newly added hardware resources taking effect since the setup time is unavoidable and sometimes very long (e.g., tens of seconds to minutes [21]). Predictive approaches [22] could avoid the long setup time and achieve good performance when the workload has intrinsic patterns. However, the real-world traffic for n-tier applications are naturally bursty and may vary significantly even within a short time [23], it is clearly challenging to predict whether new instances are required or not in advance of setup time. Our work complements both approaches.

Software reconfiguration for performance optimization has also been studied extensively before. For example, Maji et al.[24] explored how to reconfigure application parameters such as *MaxClients* and *KeepaliveTimeout* in Apache web server to achieve the optimal performance in cloud. Their focus is to mitigate the impact of interference from the co-hosted VMs. Zheng et al. [25] conduct research on automating the generation of configuration files for Internet services such as n-tier applications. Their objective is to eliminate various misconfiguration due to manual operations, which is different from our objective of performance optimization through online adaptation of soft resource allocations.

VII. Conclusion

In this paper we demonstrate the importance of soft resource allocations (e.g., server threads pool size or connection pool size) on n-tier application scaling management in cloud. Through extensive experiments of an n-tier application benchmark (RUBBoS), we observe that scaling only hardware resources without appropriate adaptation of soft resource allocation of each server can cause significant degradation of the overall system performance. We introduce a concurrency-aware model which is able to determine a near-optimal concurrency setting to each tier based on a combination of operational queuing laws and online analysis of fine-grained measurement data. We design a dynamic concurrency management framework DCM which integrates the concurrency-aware model into n-tier system scaling management. Our experiments using the real-world bursty workload demonstrate that DCM can achieve significantly more stable performance compared to hardware-only scaling solutions.

Acknowledgment

This research has been partially funded by National Science Foundation by CISE's CNS (1566443) and Louisiana Board of Regents under grant LEQSF(2015-18)-RD-A-11. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or other funding agencies and companies mentioned above.

References

[1] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab *et al.*, "Scaling memcache at facebook," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 385–398.

[2] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 1. ACM, 2012, pp. 53–64.

[3] 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.

[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] 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.

[13] 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.

[14] 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.

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

[16] 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.

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

[18] 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.

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

[20] 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)*. IEEE, 2005, pp. 217–228.

[21] 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.

[22] 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.

[23] 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*. ACM, 2009, pp. 149–158.

[24] 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.

[25] 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.