Autonomic Mix-Aware Provisioning for Non-Stationary Data Center Workloads

Rahul Singh, Upendra Sharma, Emmanuel Cecchet, Prashant Shenoy
Department of Computer Science,
University of Massachusetts Amherst
{rahul,upendra,cecchet,shenoy}@cs.umass.edu

January 31, 2011

Abstract

Online Internet applications see dynamic workloads that fluctuate over multiple time scales. This paper argues that the non-stationarity in Internet application workloads, which causes the request mix to change over time, can have a significant impact on the overall processing demands imposed on data center servers. We propose a novel mix-aware dynamic provisioning technique that handles both the non-stationarity in the workload as well as changes in request volumes when allocating server capacity in Internet data centers. Our technique employs the k-means clustering algorithm to automatically determine the workload mix and a queuing model to predict the server capacity for a given workload mix. We implement a prototype provisioning system that incorporates our technique and experimentally evaluate its efficacy on a laboratory Linux data center running the TPC-W web benchmark. Our results show that our k-means clustering technique accurately captures workload mix changes in Internet applications. We also demonstrate that mix-aware dynamic provisioning eliminates SLA violations due to under-provisioning with non-stationary web workloads, and that it offers a better resource usage by reducing over-provisioning when compared to a baseline provisioning approach that only reacts to workload volume changes. We also present a case study of our provisioning approach on Amazon's EC2 cloud platform.

1 Introduction

Online Internet applications have become popular in a variety of domains such as e-retail, online banking, finance, news, social networking and communication. Many Internet applications run on a hosting platform such as Amazon's Elastic Computing Cloud (EC2) or the Google App Engine. A hosting platform (or a compute cloud) comprises of server and storage farms housed in one or more data centers; the platform rents server and storage resources to each hosted applications and in return provides guarantees on the capacity and performance seen by each application.

Numerous studies have shown that workloads seen by Internet applications can be highly dynamic with variations at multiple time-scales [8, 13]; such variations include seasonal fluctuations such as time-of-day and month-of-the-year effects as well as sudden workload spikes caused by flash-crowds (e.g., the so-called 'slashdot effect'). To ensure a minimum level of performance and meet the contracted service level agreements (SLAs), the underlying platform will need to dynamically match allocated capacity to such observed workload changes. While long-term variations can be handled by manually allocating server capacity to an application, short-term fluctuations require an ability to dynamically provision capacity to react to workload changes in an agile and timely manner. Recently a number of dynamic capacity provisioning techniques have been proposed in the literature [23, 24, **?**, 4]. Most of these techniques track or predict changes in the workload [23, 24] and then provision sufficient capacity to handle this expected volume of requests. For instance, if a flash crowd causes the workload volume to double, so must the provisioned server capacity. Similarly commercial cloud platforms such as Amazon's EC2 only support a simple provisioning approach that adds a new server if the utilization of the current servers exceed a threshold.

In this paper, we argue that provisioning server capacity based on the expected volume of requests may not be sufficient to accurately capture the service demands of a web application. In particular it has been observed that in real production applications ranging from enterprise applications to large e-commerce sites, workload is higly variable and the request mix exhibits nonstationarity [20, **?**]. Consequently, a provisioning technique should consider both the mix of requests as well as their volumes to accurately estimate the capacity needed by an application. For example, normally infrequent heavy-tailed requests may become more frequent during a workload surge, necessitating significantly more capacity to handle these "heavy-hitters". We argue next using illustrative examples why the workload mix matters when provisioning server capacity.

1.1 The Case for Mix Aware Provisioning: Why Workload Mix Matters?

Workload fluctuations seen by Internet applications can be caused by changes in the *volume* of incoming requests or by changes in their *mix*; frequently, workload fluctuations are accompanied by both types of changes. It is clear that a significant change in the request volume requires changes in the provisioned capacity—for example, if the request rate doubles, the application needs twice as much server capacity. A change in the workload mix (i.e., the relative frequencies of different request types) is a more subtle phenomenon that has an equally important impact on capacity provisioning; however, the impact of such *non-stationary workloads* has not been adequately addressed in the research literature. We argue why the workload mix matters using two illustrative examples.

Example 1: Consider a web application that services two types of requests: *long* and *short*. Assume that short requests require 1ms of processing time and long requests take 90ms. Initially, the application receives 90 short and 10 long requests each second (total of 100 req/s). This imposes $90 \cdot 1 + 10 \cdot 90 = 990ms$ of server processing overhead in each second. Next assume that the workload mix changes to 10 short and 90 long requests/second (the total request volume is unchanged at 100 req/s). This new mix requires $10 \cdot 1 + 90 \cdot 90 = 3610ms$ of server processing time each second. Thus, the total processing demand on the server has *quadrupled* even though the incoming *request rate is unchanged*.

A provisioning technique that tracks aggregate request volumes would not notice any change in the request rate, and thus, would be unable to react to this large change in the server load. Next we explain why a provisioning technique that makes decision solely on the basis of request volumes can incur errors by over- or under-provisioning server capacity.

Example 2: Consider the above web application that services long and short requests, where short requests require 1ms and long requests require 90ms of server processing. Assume an initial request rate of 100 req/s with equal number of long and short requests (50 request each per second). Next assume that the workload doubles to 200 req/s. If the workload mix were unchanged, this would imply a doubling of the processing demand on the server, requiring twice as much server capacity. However, if the relative mix of long and short request changes when the workload doubles to 200 requests/s, then the actual demand on the server can be *greater* or *smaller* than the 2x increase in request volume. In particular, if the mix changes to 150 long and 50 short request (total of 200), this requires an increase of $\frac{150*90+50*1}{50*90+50*1} = 2.97$, which is a *three-fold* increase in server capacity for a *doubling* of request workload. Similarly, if the mix changes to 150 short and 50 long requests (total of 200 req/s), this requires an increase of $\frac{150*1+50*90}{50*90+50*1} = 1.02$ — a *mere 2% increase* in server capacity even when the workload *doubles*. Thus, a provisioning technique that naively doubles server capacity due to a doubling of request workload can under- or over-provision capacity by large amounts by ignoring the changes to the workload mix.

The above examples illustrate why the workload mix matters when determining the server capacity needed to service a particular workload. This paper presents such a *mix-aware* provisioning technique that can automatically increase or decrease the server capacity allocated to an Internet application in response to significant changes in the workload mix or the workload volume.

1.2 Research Contributions

Our autonomic mix-aware provisioning technique is designed to capture the effects of non-stationarity in Internet workloads—which manifest as changes in the observed request mix as well as changes in the volume of requests seen by servers. In order to be mix-aware, our technique first characterizes the workload mix. Instead of doing a static a priori analysis of the workload, we present an automated approach based on the k-means clustering algorithm to automatically determine the different “types” of requests present in the workload. Our technique pays particular attention to the effects of large requests seen in heavy-tailed workloads. This characterization is then used to drive a queuing theoretic model of data center applications, which computes the new server capacity needed to service the predicted workload mix. The technique then dynamically increases or decreases the number of servers allocated to the application based on the predicted capacity.

We have implemented a prototype provisioning system that incorporates our techniques on a laboratory Linux data center. We have conducted a detailed experimental evaluation of our approach by subjecting the TPC-W application to non-stationary time-varying workloads. Our results show our k-means clustering approach can accurately capture workload mix changes seen by Internet applications. We also find that our mix-aware dynamic provisioning system eliminates SLA violations due to under-provisioning with non-stationary web workloads, while also reducing over-provisioning and achieving better resource usage when compared to baseline provisioning techniques that react to workload volume changes.

The rest of this paper is organized as follows: Section 2 gives a system overview. Our clustering technique and mix-aware provisioning policy are described in sections 3 and 4, respectively. Implementation details are discussed in Section 5 and Section 6 shows experimental results. Section 7 presents related work and we conclude in Section 8.

2 System Overview

This section presents the model of the hosting platform and Internet applications assumed in our work as well as the architecture of our dynamic provisioning system.

2.1 Data Center Model

We assume that web applications run on a hosting platform (or the hosting "cloud") that comprises a data center with a cluster of commodity servers that are interconnected by a high-speed LAN (Gb or 10Gb Ethernet); one or more high bandwidth links connect the hosting platform to the Internet.

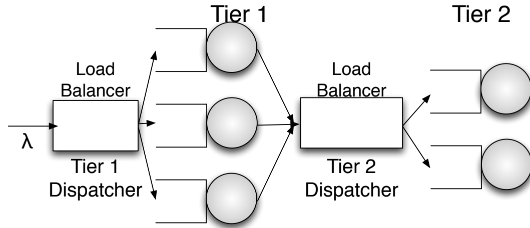


Figure 1: Multi-tier application model

The platform is assumed to host multiple Internet applications. Modern Internet applications are distributed with several components; a multi-tier architecture is a popular technique for constructing such applications. In this architecture, each application tier provides a certain functionality, and the various tiers form a processing pipeline. Each tier receives partially processed requests from the previous tier and feeds these requests into the next tier after local processing (see Figure 1). For example, an online bookstore can be designed using two tiers—a web tier that receives client connections and implements the application logic, and a backend database that stores catalogs and user orders.

The various tiers of the application are assumed to be distributed across different servers of the data center. Depending on the desired capacity, a tier may also be *replicated* via clustering. In an online bookstore, for example, the web tier can be a clustered Apache Tomcat server that runs on multiple machines. Such replication enables the tier capacity to be scaled in proportion to the number of replicas. Each clustered tier is also assumed to employ a load balancing element that is responsible for distributing requests to replicas in that tier. If a tier is both clustered and replicable on-demand, it is assumed that the number of servers allocated to it, and thus the provisioned capacity, can be varied dynamically.

The goal of the hosting platform is to monitor the workload seen by each tier and dynamically add or remove replicas at each tier to match the incoming workload (while striving to meet the application's desired response time SLA). The SLA is assumed to be specified in terms of a high percentile of the response time distribution (e.g., 95 percentile of requests should have a response time less than 1 second).

2.2 Provisioning System Overview

The architecture of our dynamic provisioning system is depicted in Figure 2. The heart of the technique lies in the provisioning logic which runs on a control node of the data center. It consists of an *aggregator* that monitors the incoming workload at each tier and gathers workload statistics by processing the request logs produced by each component. For new applications, these statistics are used by the *mix determiner* for one-time analysis to characterize the application workload. This characterization is used by the workload predictor to estimate the future workload mix and then fed into the queuing model to compute the server capacity needed at each tier. The *actuator* can then provision additional servers from the free pool or return unneeded servers back to the pool.

The following sections describe (i) how the mix determiner employs an automated clustering-based approach for determining the request types seen by an application, and (ii) how this workload characterization can then be employed to design a mix-aware provisioning technique using a queuing theoretic model.

3 Characterizing Workload Mixes with Clustering

To characterize a workload mix, requests which have similar service demands are grouped into a service class. The service demand of a request is measured by the service time of the request. Service classes are used to determine how much capacity to provision. Service classes can be determined manually but this is a long, error prone and tedious process. We propose a technique to perform this classification automatically.

Our approach is based on the well-known *k-means* clustering algorithm. *k-means* clusters n objects into k partitions ($k < n$) by minimizing total intra-cluster variance to find the centers of natural clusters in the data. The algorithm assumes that the variance is an appropriate measure of cluster scatter. Other clustering algorithms could be used for this purpose, but *k-means* has the advantage that it is well studied and is known to converge very quickly in practice. The vanilla *k-means* algorithm requires k (the number of clusters) as an input; in our case, however, the number of request classes in *not known* a priori. Hence, we must use a modified approach that first determines the appropriate k for a particular workload and then clusters the workload into k different request classes.

Our technique for doing so consists of first collecting unique request service times to partition them into clusters. As explained in section 1.1, infrequent queries might have a long service time which translates into a heavy-tailed workload. We adjust clustering of such workloads by splitting the large tail cluster into smaller clusters less sensitive to service time variations of larger queries. Finally, we observe the frequency of the requests within each each cluster to find the cluster means.

We now detail the 3 steps of our enhanced clustering algorithm:

Step 1 - Determine k and partition unique request types

into clusters: We use an iterative approach to determine the the best k value for a particular application. The idea is to iteratively run *k-means* with every value of k and compute 4 variables: coefficient of variation[1] of intra-cluster distance, coefficient of variation of inter-cluster distance, ratio of intra-cluster variance to inter-cluster variance, and ratio of intra-cluster coefficient of variation to inter-cluster coefficient of variation. The best k value minimizes intra-cluster variance and maximizes inter-cluster variance [14]. Determining the best k value is a one-time activity for any given application.

Each cluster represents a service time class defined by a lower and upper service time bound. As we use the entire set of *unique* service times of the application, any request at runtime will fall within a cluster.

Step 2 - Adjust for heavy-tailed workloads: It is common for service time distributions to have a long tail due to infrequent heavy requests. *k-means* is likely to aggregate all these infrequent service times in a single large cluster.

Example 3: Consider a cluster that contains 2 requests of 500ms service time and 1 request of 2000ms. Then the average service demand for the cluster is $\frac{2*500+2000}{3} =$ 1000 ms. If the workload changes and invokes one 500ms and two 2000ms requests, the new cluster average becomes $\frac{500+2*2000}{3} = 1500$ ms, which is a large shift in the cluster mean.

To prevent such huge shifts in cluster averages, we divide large clusters into smaller clusters of fixed size. To do so, we specify a threshold *max size* on the maximum size of a cluster; if a cluster containing the tail of the request exceeds this size, it is broken up into multiple small clusters, each of size *max size*.

Step 3 - Compute cluster means: We determine the mean for each of the k clusters by using the set of all services times seen in an interval as opposed to the unique request service time used in previous steps. A cluster mean is the average service demand for all requests falling in that cluster. If a cluster has the service times $S = \{s_1, s_2, s_3, \ldots, s_k\}$ which appear with frequencies $F = \{f_1, f_2, \ldots, f_k\}$, the cluster mean is given by $\frac{\sum_{i=1}^{i=k} s_i f_i}{\sum_{j=1}^{j=k} f_j}$.

Step 4 - Recomputation of cluster means: As the workload mix changes over time, the frequency of different request types will change, causing the cluster mean to change. In a mix-oblivious scenario, all service times belong to one cluster. The cluster mean changes very frequently, and by large amounts, as the workload mix changes. With an optimal number of clusters, however, the cluster means remain more stable over time even in the presence of workload changes.

Our experimental evaluation in section 6.2 shows that cluster means changes do not exceed 5% in case of workload changes. Therefore re-estimating cluster means is not a frequent operation. This re-estimation of cluster means, while infrequent, can be done efficiently since the number of clusters is constant—we simply scan through all clusters and find out the frequency of each request type in each cluster to re-calculate the cluster mean.

[1]coefficient of variation or variation coefficient is defined as a ratio of the standard deviation to the mean, i.e. $c = \sigma/\mu$;

4 Autonomic Mix-Aware Provisioning

A dynamic provisioning algorithm must decide how many servers to allocate at each tier of an application so that a specified Service Level Agreement (SLA) is not violated. The provisioning decision for each tier of a multi-tier application is taken independently. Once the provisioning logic decides the number of servers required for each tier, the configuration is updated accordingly by an actuator. Figure 2 gives an overview of the architecture of the system.

The key insight behind our provisioning approach is to not only consider changes in workload volume (like in a volume-based provisioning approach [23]) but also track changes in the workload mix using service time clusters determined in section 3. Our algorithm continuously monitors the workload to make two decisions: (i) when to trigger provisioning, and (ii) how much additional capacity to provision. We discuss each decision in turn.

4.1 When to provision?

The effectiveness of provisioning decisions relies on invoking the provisioning algorithm at pertinent points in time. Our approach uses three different types of triggers:

Periodic: the provisioning logic is invoked at regular time intervals.

Triggered by change in volume: a drastic change in the volume of requests triggers provisioning.

Triggered by change in the mix: a change in the workload mix can be used to invoke provisioning to respond to these changes.

By default, our approach invokes the provisioning algorithm periodically to ensure that provisioned capacity is sufficient to service the expected workload. The periodicity of invocation can be decided by the administrator depending on the datacenter; e.g. in the case of EC2 it can be less than 1 hour since the billing is done every hour, while in the case of a private data-center it can be more. Besides this, any sudden changes in the workload volume or the workload mix beyond a threshold also serve as additional triggers, since either can substantially change the service demand of the application.

4.2 How much to provision?

Once the provisioning algorithm has been triggered, it must then provision additional capacity, if needed, for each tier of the application so that its SLAs are met. We assume that SLAs in a multi-tier application are specified as end-to-end response time perceived by the client. For instance, an SLA can specify that the mean of the end-to-end response time must be less than a value or that the 95th percentile of response time should be less than a threshold. The constraint

on the end-to-end response time is suitably broken down into constraints on *per-tier* response times. This way, each tier can be treated independently of each other. In this paper, we use SLAs specified in terms of the 95th percentile of the end-to-end response time. Given these per-tier SLAs, provisioning the correct number of servers for each tier is a three step process:

Step 1 - Estimate λ_i for each cluster: The clustering algorithm described in section 3 divides the entire workload into clusters. A predictor for each cluster forecasts λ_i the arrival rate of requests in that cluster. Predictors based on time series can be used to provide accurate results for data center workloads [8],[23]. The total volume of requests, λ_{total}, is obtained by summing the λ_i's of each cluster.

Step 2 - Queuing model to predict capacity: We model multi-tier applications using a network of queues. Each tier is represented as a queue. Requests coming into a tier are modeled as requests visiting a queue. The queuing discipline is assumed to be first-come-first-serve. The requests wait in the queue to be serviced, and once serviced they move to the next queue in the network. Each queue is modeled as a G/G/1 queuing system. This system can handle an arbitrary distribution of arrivals and an arbitrary distribution of service times.

We use Kingman's approximation for the waiting time in a G/G/1 queue in the heavy-traffic case [11]. This result gives an approximation of the waiting time distribution when the utilization $\rho \cong 1$ (but remains strictly less than one so that the system is still stable). The probability distribution function of the waiting time in the queue is exponentially distributed with mean $\frac{{\sigma_a}^2+{\sigma_b}^2}{2(\frac{1}{\lambda}-\bar{x})}$ where ${\sigma_a}^2$ is the inter-arrival time variance, ${\sigma_b}^2$ the service time variance, λ the request arrival rate, and $\bar{x}$ the average service time.

In our provisioning experiments the service level agreement is defined in terms of the 95th percentile response time. Since the waiting time distribution is exponential, the 95th percentile of the waiting time, $\alpha_W(95)$, can be expressed in terms of the mean waiting time $E[W]$. If the SLA requires the 95th percentile response time to be less than y seconds, the maximum arrival rate of requests that can be sustained by one server is given by:

$$\lambda < \left[\bar{x} + \frac{{\sigma_a}^2 + {\sigma_b}^2}{2\,(y/3 - \bar{x})}\right]^{-1} \quad (1)$$

The average service time is $\bar{x} = \frac{\sum_{i=1}^{i=k} \lambda_i d_i}{\sum_{i=1}^{i=k} \lambda_i}$ where $\lambda_i =$ arrival rate in cluster i as found in step 1, and $d_i =$ cluster mean of cluster i. The values of the inter-arrival variance and service times ${\sigma_a}^2$ and ${\sigma_b}^2$ are obtained using online monitoring of the tier. We substitute these values in Equation 1 to find out λ. This gives the capacity of one server. Using λ_{total} calculated in step 1, we find the number of servers required at this tier:

$$N = \left\lceil \frac{\lambda_{total}}{\lambda} \right\rceil \quad (2)$$

Our mix-aware provisioning algorithm is formally described in Algorithm 1.

Input: Let there be k clusters at this tier, incoming volume of requests in each cluster $P = \{\lambda_1, \lambda_2, \lambda_3, \ldots, \lambda_k\}$, the cluster mean of each cluster $D = \{d_1, d_2, d_3, \ldots, d_k\}$, the 95th percentile response time threshold for this tier y, the variance of inter-arrival time ${\sigma_a}^2$, the variance of service time ${\sigma_b}^2$

Output: Number of servers needed for this tier

$\lambda_{total} = \sum_{i=1}^{i=k} \lambda_i$

$\bar{x} = \frac{\sum_{i=1}^{i=k} \lambda_i d_i}{\sum_{i=1}^{i=k} \lambda_i}$

$\lambda_{per-server} = \left[\bar{x} + \frac{{\sigma_a}^2+{\sigma_b}^2}{2(y/3-\bar{x})}\right]^{-1}$

$N = \left\lceil \frac{\lambda_{total}}{\lambda_{per-server}} \right\rceil$

return N

Algorithm 1: Find number of servers required at a particular tier

Step 3 - Applying the new configuration: Once the new number of servers is determined for each tier, an actuator adds or removes servers at each tier to achieve the desired configuration. New added servers are taken from a free pool, and removed servers are returned to the free pool.

Note that heterogeneous platforms can be handled by using simple multiplication factors between servers depending on their hardware characteristics. For example, Amazon EC2 provides various instances (small, medium, large, xlarge...) each with different hardware resources. By running the web application on each instance, it is easy to determine a performance factor like, for example, a *medium* instance performs 1.5 times better than a *small* instance, or a *large* instance provides 2x the throughput of a *small* instance. The number of servers can then be expressed in number of *small* instances and the actuator can realize the configuration to apply using a mix of different instances. For example, using the performance factors described previously, adding 5 *small* servers could be realized by adding 2 *medium* and 1 *large* servers or 2 *large* and 1 *small* servers.

5 Prototype Implementation

We have built a prototype provisioning system that incorporates our mix-aware provisioning technique. Our prototype assumes a cluster of Linux servers in a data center that are interconnected by a high-speed LAN such as a gigabit ethernet. We assume that one node of the data center is the control node that runs our provisioning logic. Our prototype assumes that each tier of a multi-tier application runs on one or more servers and that the tier can be replicated dynamically to scale its capacity. Figure 2 gives the architectural overview of the different components of our provisioning system.

Servers: The data center hosts a multi-tier application and servers are allocated in 4 pools: web tier, database tier, load

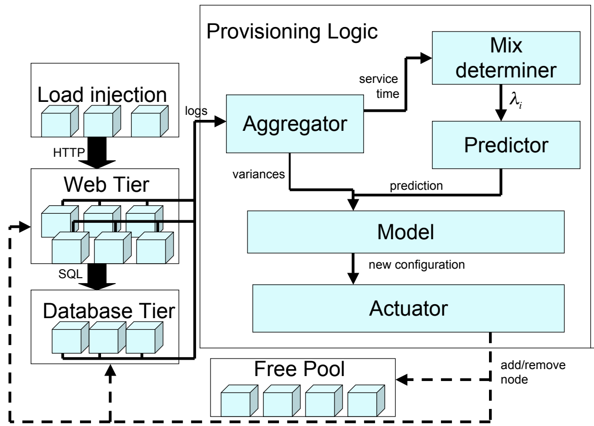


Figure 2: Provisioning architecture overview

injection and free servers. When the provisioning logic decides to add a node to the web or database tier, a server is picked from the free server pool. Unused nodes freed by the provisioning decisions are returned to the free pool. The load injection server pool is used to emulate users.

The web tier uses Apache Tomcat 5.5.26 application server and the database tier is MySQL 5.1.26 using MySQL master/slave replication. We have instrumented the Tomcat source code to report the request service time. To get accurate service times we use the Linux 2.6.26 *getrusage* system call that captures the time consumed by a particular thread. We also configured Tomcat to log the arrival time and response time of each request. To measure the variance of the inter-arrival time, we use finer granularity timers to get sub-milliseconds accuracy. The logs are stored on a shared filesystem. MySQL request service time is determined offline by measuring the query execution time of each query in isolation on a standalone system. The query log is activated to record query arrival time. The above modifications can be done in an application agnostic manner and is thus generic for any web-application deployed on Tomcat and MySQL.

The provisioning logic contains 5 components: aggregator, mix determiner, predictor, model and actuator. They are all implemented in Java; we have implemented the inter-component communication using RMI.

Aggregator: The *aggregator* collects information from all Tomcat and MySQL servers at regular time intervals via the shared filesystem. Tomcat logs are parsed to find the i) ${\sigma_a}^2$ the variance of inter-arrival time, ii) ${\sigma_b}^2$ the variance of service-time, iii) the average response time and iv) the set of service times seen in the last interval. For MySQL, the *aggregator* gets the SQL queries from the request log file. It then looks up the service time for each query from the numbers collected during offline profiling. The statistics for each tier are calculated by aggregating the corresponding statistics of all the replicas of the concerned tier.

Mix-Determiner: The *mix-determiner* collects the service time values observed at each tier from the *aggregator*. Service times at a tier are distributed into the tier clusters. This way the *mix-determiner* finds out the arrival rate of each cluster (i.e. λ_i). It also computes the current cluster mean. If the current cluster means deviate more than 0.5 times the standard deviation of any cluster, *mix-determiner* automatically triggers the process of re-estimation of new clusters means.

Predictor: The *predictor* takes the arrivals of each bin (λ_i) as an input from the *mix-determiner*. In the current implementation, we directly use the arrival rate values from the load injectors. Predictions are sent to the *model*.

Model: The *model* collects statistics from the *aggregator* and the *predictor*. It processes them to find out the number of servers needed at each tier by using algorithm 1. The results are sent to the *actuator* to implement the new system configuration.

Actuator: The *actuator* reconfigures the system to provide the exact capacity that has been decided by the *model*. It adjusts the number of servers at each tier accordingly by either adding new servers from the free pool, or removing servers and returning them to the free pool. Servers returned to free pool are handled in a datacenter specific manner so as to increase benefit of the reconfigured system; for e.g. in the case of a data center with dedicated servers, actuator suspends the machines to save power and wake them up when needed, while in the case of EC2 it detaches the compute infrastructure to save the rental cost. The *actuator* adds servers to the application in a tier specific manner, i.e. new MySQL nodes are always added as slave nodes and the master node is never removed, while Tomcat nodes, all being identical, are added or removed indifferently.

6 Experimental Evaluation

We have tested our proposed clustering and provisioning techniques using a standard multi-tier application benchmark. We describe our experimental setup in section 6.1. As the provisioning policies apply independently to each tier, we have chosen for space constraints to only present our clustering results for the database tier (section 6.2) and provisioning results for the web tier (section 6.3). Finally, we present a case study on Amazon's EC2 public cloud platform in section 6.4.

6.1 Experimental Setup

We use the ObjectWeb [16] implementation of the TPC-W benchmark [22] as our reference multi-tier application. TPC-W models an online retail bookstore. The TPC-W specification describes 14 different web interactions and three different workload mixes: browsing, shopping and ordering. These mixes differ in the relative frequency of each interaction which translates in different amount of read-only and

update transactions. The browsing mix workload has 5% updates, the shopping mix 20% updates and the ordering mix 50% updates.

The users interacting with the web site are emulated by the RBE (Remote Browser Emulator) that browses the web site according to the workload definition. For some provisioning experiments, we have also used the *httperf* [15] load generation tool. *Httperf* allows us to generate variations in the workload that are different from the one hard-coded in the RBE.

We have built a data center prototype to evaluate the effectiveness of our clustering and provisioning techniques. We use a total of 16 physical servers each with an Intel Pentium4 2.40GHz with 2GB RAM interconnected by a Gigabit Ethernet network. One server is dedicated to run the provisioning logic. The rest of the servers are used to host the different components of the TPC-W web benchmark. Each server runs a linux kernel version 2.6.26.

We also ran additional experiments on the Amazon EC2 [2] public cloud platform (section 6.4) where we use small server instances for the application tiers and a dedicated instance to run the provisioning logic. Each server runs a linux kernel version 2.6.21 with the Xen [3] virtualization extensions.

6.2 Clustering Evaluation

This section presents the results of our clustering experiments for the TPC-W benchmark with the service times observed at the database tier.

6.2.1 Initial clustering

As described in section 3, the first step is to find out the set of unique service times for each tier. We run each workload mix of the TPC-W benchmark with a constant number of 800 emulated users to collect all possible service times in all mixes. We use offline profiling to get MySQL service times as described in section 4.

To find out the right number of clusters (or service classes) in the service time dataset, we employ the technique described in section 3. We try multiple values of k, the number of clusters in the data, starting with 3 up to 14 (the total number of interactions in TPC-W). For each k, we run *k-means* and compute 4 variables [14]: 1) inter-cluster coefficient of variation (CV_{inter}), 2) intra-cluster coefficient of variation (CV_{intra}), 3) ratio between intra and inter-cluster variance (β_{var}) and 4) ratio between intra and inter-cluster coefficient of variation (β_{cv}). We then choose the value of k that gives the smallest values of β_{cv} and β_{var}.

Figure 3(a) shows CV_{inter}, CV_{intra} and β_{cv} with increasing values of k. Both β_{cv} and CV_{intra} reach their lowest values when $k = 5$ which is the optimal cluster size for our TPC-W setup. Clustering performance degrades sharply with significantly higher values of β_{cv} and CV_{intra} with $k > 5$. CV_{inter} remains stable with an increasing number of clusters. Figure 3(b) shows how β_{var} changes with an increasing number of clusters (k). β_{var} reaches its smallest value for $k = 5$ with cluster centers [{0,0.125}, {0.132, 0.279}, {0.279, 0.343 } { 0.343, 0.396} { 0.396, 0.57 }] (each value is the range of service time in seconds for that cluster). Higher number of clusters gives higher β_{var} values. These 2 figures indicate that the appropriate number of clusters is 5 for the database tier.

The results of *k-means* with 5 clusters shows that the high service time clusters are mostly composed of just two SQL queries. These two queries are the queries generated by the "Best Sellers" servlet and the "Admin Confirm" servlet. The authors of [5] also found these 2 interactions to require more table joins than the others and to be the most intensive on the DB tier. For this particular workload, *k-means* distributes the heavy queries across multiple clusters and does not require further tail adjustment.

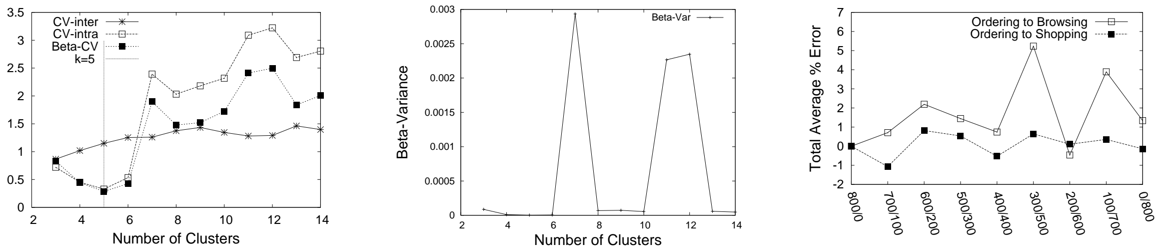
6.2.2 Impact of non-stationary workloads

In Section 3, we discussed variations in cluster means. We conducted two experiments with different non-stationary workloads to show that cluster average variation is limited even with significant workload mix changes.

In a first experiment, we change the workload from ordering mix to browsing mix. Initially there are 800 sessions performing interactions defined by the ordering mix. We gradually replace ordering mix sessions with browsing mix sessions by steps of 100 so that there are always 800 sessions. The average service time at the database tier is found out by offline profiling. Also the service demand is estimated by keeping the cluster average d_i fixed for each cluster. The service times found in an interval are then put into the clusters to find λ_i for each cluster. We then find the average service time $\bar{x}$ as shown in Algorithm 1. In a second experiment, we transition the workload from 800 ordering mix sessions to 800 shopping mix sessions.

Figure 3(c) shows the evolution of errors in the service demand estimates for both workload transitions. The x-axis represents the number of ordering mix sessions decreasing from 800 to 0. Depending on the experiment, the number of browsing and shopping sessions are increasing by 100 each time the ordering sessions decrease by 100. The maximum error observed was 5% for the transition from ordering to browsing. The error remains below 1% for the transition from ordering to shopping.

The lower errors observed in the second experiment are due to the fact that the shopping and ordering mix have a closer mix of interactions than the ordering and browsing mix. The error can be reduced by periodically re-estimating the cluster averages from past data. These experiments confirm that recomputing cluster averages does not need to be done frequently. The service demand estimates remain accurate even in the presence of a major workload change.



(a) Inter and intra cluster coefficients of variation and β_{cv} as a function of the number of clusters

(b) β_{var} as a function of number of clusters

(c) Error in estimate of service demand with non-stationary workloads

Figure 3: Clustering Experiments

6.3 Provisioning Experimental Evaluation

In this section, we evaluate the accuracy of our provisioning technique with non-stationary workloads. As provisioning decisions are applied independently at each tier, these experiments are focusing on the web tier only. The application is assumed to require an SLA where 95th percentile of the web-tier response time must not be greater than 2 seconds. The database tier is over-provisioned with 4 replicated instances of MySQL to ensure that the database is not a bottleneck resource.

We compare the standard provisioning policy based only on volume of requests [23] with our technique that adds mix-awareness. In both cases, the algorithms are provided with the request arrival rate from the load injectors (λ_i and λ_{total}). When new Tomcat servers are added to the web tier, the load injectors are configured to distribute the sessions evenly among Tomcat instances to ensure a good load balancing.

6.3.1 Non-stationary workload with constant volume

This first experiment shows the impact of non-stationary workloads on provisioning decisions when the number of sessions remains constant. We expect mix-aware techniques to improve over techniques that will not detect changes in the constant arrival rate of requests.

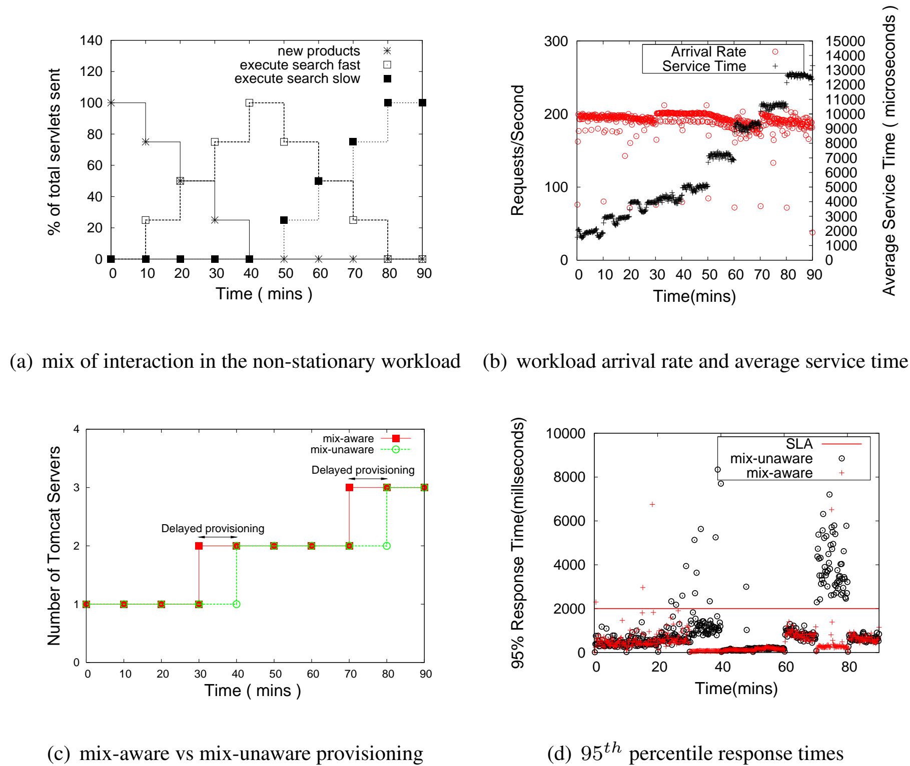
The Workload: We have built a workload of a constant number of 1000 sessions with a different mix of 3 different servlets. We use 3 interactions of the TPC-W benchmark: *"new products"* and 2 versions of *"execute search"*. As the service time of the *"execute search"* servlet depends on the complexity of the term searched, we have created 2 queries that generate a short and a long service time that we call *"execute search fast"* and *"execute search slow"*, respectively. The measured service times on our prototype data center are 2.6ms for *"new products"*, 5.5ms for *"execute search fast"* and 14.0ms for *"execute search slow"*

Figure 4(a) shows the combination of requests that is sent to the TPC-W application over a 90 minutes experiment. We use httperf to send requests with a 5 seconds think time between 2 interactions. In the first 50 minutes, the percentage of *"new products"* requests decreases while the percentage of *"execute search fast"* requests increases. This causes the average service time of requests on the web tier to increase. Similarly, in the next 40 minutes, *"execute search fast"* requests are transitioned to *"execute search slow"* requests, leading to an even further increase in the average service time. Figure 4(b) shows the constant average arrival rate of requests, and the average service time of requests that increases at the web tier during the experiment.

Provisioning Decisions: We compare our mix aware provisioning scheme with a provisioning scheme that looks only at the volume of requests. The provisioning logic is invoked every 10 minutes. Figure 4(c) shows the decisions taken by the 2 provisioning policies in terms of the number of servers allocated to the web tier during this experiment. The figure shows that the *mix-aware* provisioning policy is able to anticipate faster capacity requirements than the *mix-unaware* on two occasions. The mix-aware policy decides to allocate a new Tomcat server to the web tier at t=30 and t=70 minutes. By predicting that the mix of requests about to come is changing, the mix-aware provisioning anticipates the capacity needs. The mix-unaware policy however, does not see changes in the volume of requests and assumes that it will continue to see the past service demand in the future. It is only when a response time increase is observed at a later iteration that a new server is added to the web tier.

Response Times: Figure 4(d) shows the 95th percentile response times as seen by the clients while running the two provisioning approaches. These figures show a comparison between the two policies in terms of how much they violate the SLA. Note that with the *mix-unaware policy* the 95th percentile crosses the 2 seconds line on many occasions after the mis-predictions at t=30 minutes and t=70 minutes. The *mix-aware policy* anticipated the workload changes and reduced the SLA violations by 94%.

Result: This experiment illustrates that when the volume of requests is constant the resource demand on a tier may still change because of a change in the workload mix. A mix-aware provisioning is therefore able to provision resources



(a) mix of interaction in the non-stationary workload

(b) workload arrival rate and average service time

(c) mix-aware vs mix-unaware provisioning

(d) 95th percentile response times

Figure 4: Non-stationary workload experiment with a constant arrival rate

to account for these changes and avoid SLA violations that would be experienced with a provisioning scheme only looking at the volume of requests.

6.3.2 Non-stationary workloads with a varying volume

In this second experiment, we combine a varying workload a change in the arrival rate.

The Workload: The workload mix is the same as the experiment described in section 6.3.1 until t=60 minutes, then we transition the mix back symmetrically to its original composition for a total of a 120 minutes experiment. To vary the arrival rate, we increase the number of sessions from 1000 to 2000 at time t=30 minutes.

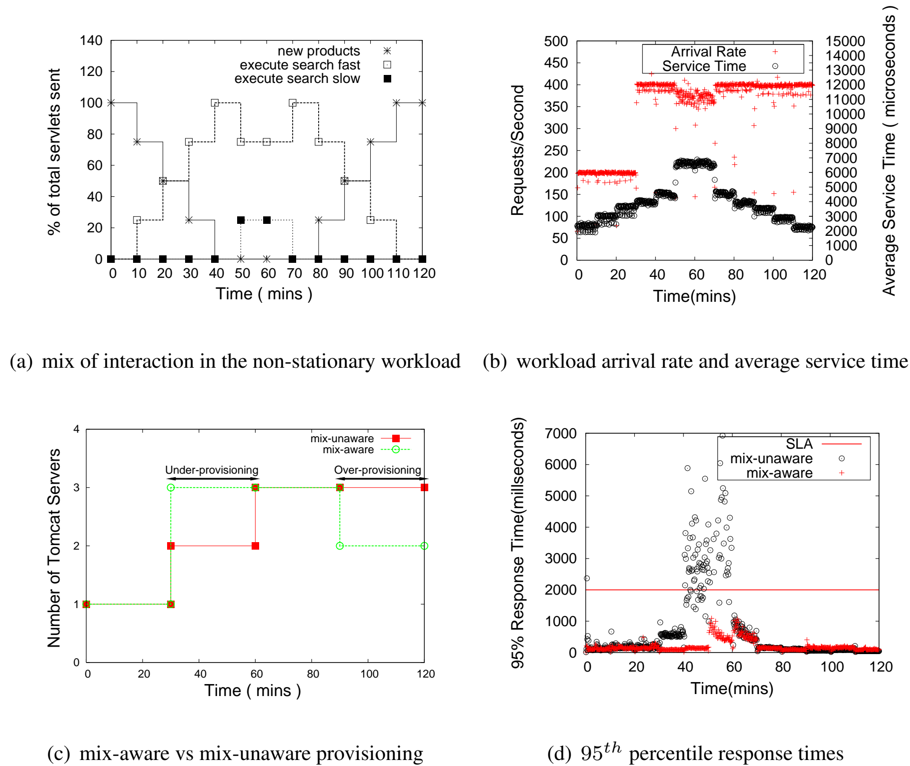
Figure 5(a) depicts the workload mix and Figure 5(b) shows the requests arrival rate and the average service time at the web tier during this experiment. Due to the nature of the mix, the average service time keeps increasing for the first 60 minutes and then decreases for the last 60 minutes.

Provisioning Decisions: The provisioning logic is invoked every 30 minutes. Figure 5(c) shows the decisions taken by the mix-aware and mix-unaware provisioning policies. At t=30 minutes, the number of sessions has doubled. The mix-unaware provisioning policy doubles the number of servers to respond to this increase in volume of requests. The mix-aware policy is able to capture the workload mix change in addition to the increased volume of request. By taking into account the mix change, mix-aware provisioning allocates an additional server for a total of 3 servers at the web tier. The mix-unaware policy only catches up 30 minutes later at the next iteration to provision a third server.

After t=60 minutes, the workload mix changes with a decreasing service demand. At t=90 minutes, the mix-aware policy detects the mix change and decides to decrease the number of servers from 3 to 2. The mix-unaware policy however only considers the volume of requests. Since the volume is the same it assumes that it will continue to need 3 servers and leaves the system over-provisioned.

Response Times: Figure 5(d) shows the 95th percentile response times during this experiment with both provisioning policies. As expected we see that the mix-unaware policy leads to SLA violations after the misprediction at t=30 minutes. Similarly, after t=90 minutes mix-unaware response times are very low due to the over-provisioned web tier. mix-aware response time remains below SLA requirements throughout the experiment with an optimal number of servers (i.e. no SLA violations).

Result:Even when the volume of requests is changing, a mix-aware provisioning policy refines traditional predictions based on volume changes by preventing both under-provisioning and over-provisioning.



(a) mix of interaction in the non-stationary workload (b) workload arrival rate and average service time

(c) mix-aware vs mix-unaware provisioning (d) 95th percentile response times

Figure 5: Non-stationary workloads with a varying arrival rate

6.4 EC2 Case Study

In this section, we experiment our technique on the Amazon EC2 (public cloud platform). Unlike our prototype data center where our servers are dedicated to the benchmark execution, EC2 server instances are virtual machines potentially competing with other virtual machines to share the same physical resources [?]. We have calibrated the system on a single server instance and observed that the peak throughput was 15 requests/second for the browsing mix and 13 requests/second for the shopping mix with 120 clients and 80 clients, respectively.

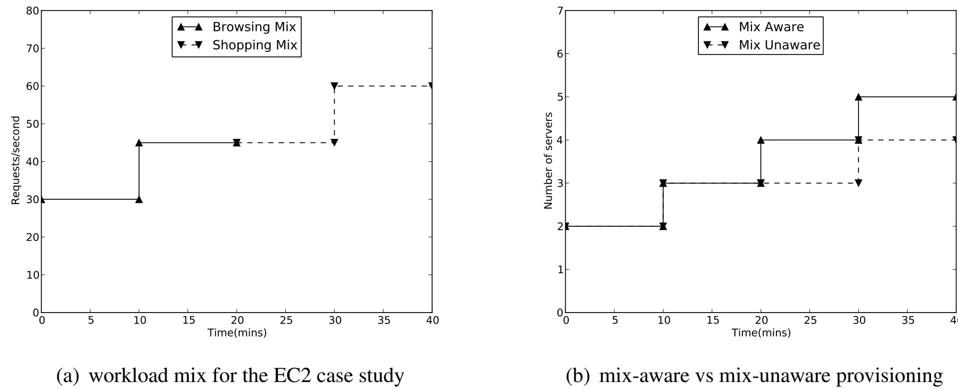
Figure 6(a) shows the TPC-W workload that we used for our experiment. We initially use a browsing mix workload of 30 requests/second that increases to 45 requests/second after 10 minutes. We then switch the workload from browsing to shopping mix at t=20 minutes using the same number of client requests. We finally increase the workload to 60 requests/second for the last 10 minutes of the experiment.

Figure 6(b) shows the decisions that are taken by the mix-aware and the mix-unaware provisioning techniques and the provisioning logic was invoked every 10 minutes. When the mix changes at t=20 minutes, the mix-aware technique provisions an additional server that is required to sustain the 45 requests/second throughput for the shopping mix. The mix-unaware remains under-provisioned as its capacity requirements are based on its observations for the browsing mix. When the throughput finally reaches 60 requests/second, the mix-aware algorithm provisions a fifth server that allows to sustain up to 65 requests/second. The mix-unaware technique provisions only 4 servers which can serve 60 requests/second of the browsing mix but only 52 requests/second of the shopping mix.

We have shown in this experiment that our mix-aware technique was effective on a public cloud platform. The response time numbers (not presented here for lack of space) are similar to the ones observed on our private cloud platform. In future works, we are planning to study the effects of virtualization on longer experiments and the use of heterogeneous server types (large and extra large EC2 server instances) in cloud platforms.

7 Related Work

A number of recent efforts have focused on dynamic capacity provisioning for Internet applications [21], [6]. In the context of system-level efforts, Shirako [10] is a system that supports on-demand leasing of network resources, including virtual machines. Shirako decouples leasing mechanisms from resource management policies and provides a flexible, extensible framework for incorporating different resource types and policies. vManage [12] is a system for coupling and coordinating energy- and resource-management in data centers dur-



(a) workload mix for the EC2 case study

(b) mix-aware vs mix-unaware provisioning

Figure 6: Non-stationary workloads on Amazon EC2

ing provisioning and placement; such coupling enables coordination of data center management policies across different layers and yields better performance and stability. Violin [18] is a system for dynamically constructing/provisioning groups of isolated virtual machines within a distributed virtualized environment; Violin focuses on network-level constructs for dynamically provisioning and isolating groups of virtual machines running components of an application.

In the context of provisioning algorithms, one class of techniques focuses on changes on the workload volume for making provisioning decisions and is mix-unaware. Our past work on provisioning includes a queuing-based approach for provisioning multi-tier applications [23]. Like the present work, the approach models each tier as a G/G/1 queue. However, the approach is mix unaware and only uses the peak session arrival rate to capture the workload and provisioning capacity to service this peak workload. Thus, the approach faces the limitations discussed in section 1.1. An alternate approach uses a G/G/N queuing model to compute the number of servers necessary to maintain a target utilization level [17]. This strategy is shown to be effective for sudden increases in request arrival rate. Other efforts have employed similar M/G/1 queuing models in conjunction with offline profiling to model service delay and predict performance [21] (we use such models for provisioning). The approach in [24] formulates the application tier server provisioning as a profit maximization problem and models application servers as M/G/1/PS queuing systems; the approach only considers the impact of different number of end-clients (and thus, request volumes) and does not specifically focus on the impact of server processing times or different mixes.

Classical feedback control theory has also been used to model the bottleneck tier for providing performance guarantees for web applications [1]. This approach focuses on web servers serving static content, where service time can be estimated from the request size. Composition of adaptive feedback systems has been studied in [9] where a co-adaptation mechanism for dealing with composition of poorly tuned feedback loops in web applications was proposed. Similarly machine learning techniques have been used for provisioning, such as the k-nearest neighbor approach to provision the database tier [6].

A few recent techniques have taken request classes (i.e., mixes) into account but the mix is assumed to be specified *a priori*. Zhang et. al. [**?**] use a multi-class model to capture the dynamics of workload by employing a fixed set of 14 predefined transactions-types and leverage it to predict the performance of a multi-tier system. Another recent effort has employed a network of queues to model a multi-tier Internet application that services multiple types of transactions. The authors employ approximate mean-value analysis (MVA) to develop an online provisioning technique but the request classes are assumed to be known a priori [4]. In contrast to these efforts, our work automates the process of characterizing the workload mix and uses this characterization to provision system capacity. Further, while most of these multi-class efforts have focused on analytic methods, our approach has involved a full prototype implementation and experiments on an actual Linux cluster.

k-means clustering is a common technique used for static workload analysis. In [7], *k-means* is used for workload analysis and demand prediction. In constrast, we use *k-means* to automatically characterize the workload and use queueing theory approaches for provisioning. Further, our focus is on designing a fully functional prototype system that has been implemented, while the focus in [7] is on the analysis of real traces. While we employ *k-means* to automatically characterize the workload, other automatic workload characterization techniques can be used for this purpose as well. For example, a recent effort has used independent component analysis (ICA) to automatically groups requests based on service demand [19]; such approaches can be also used in conjunction with our provisioning algorithm.

8 Conclusion

In this paper, we have shown that non-stationarity in application workloads, which causes request mix to change over time, have a significant impact on the overall capacity demands imposed on a data center. We have proposed a new technique based on k-means clustering to automatically determine workload mixes and a queuing model to predict server capacity for a given workload mix. We have implemented a prototype provisioning system that incorporates our mix-aware approach and evaluated it on a prototype Linux data center. Our experimental results show that k-means clustering can accurately captures workload mix changes. Our mix-aware dynamic provisioning system improves over volume-based provisioning techniques by eliminating SLA violations due to under-provisioning with non-stationary web workloads, while offering better resource usage by reducing over-provisioning. We also presented a case study of our provisioning approach on Amazon's EC2 cloud platform.

Acknowledgements: We would like to thank our anonymous reviewers and our shepherd, Jeanna Matthews, for their helpful comments. This research was supported by NSF grants CNS-0720271 and CNS-0720616

References

[1] T. F. Abdelzaher, K. G. Shin, and N. Bhatti. Performance Guarantees for Web Server End-Systems: A Control-Theoretical Approach. *IEEE Transactions on Parallel and Distributed Systems*, 13(1):80–96, 2002.

[2] Amazon elastic computing cloud (ec2). `http://aws.amazon.com/ec2`.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03), Bolton Landing, NY*, pages 164–177, October 2003.

[4] M. N. Bennani and D. A. Menasce. Resource allocation for autonomic data centers using analytic performance models. In *ICAC '05: Proceedings of the Second International Conference on Automatic Computing*, pages 229–240, Washington, DC, USA, 2005. IEEE Computer Society.

[5] H. W. Cain and R. Rajwar. An architectural evaluation of java tpc-w. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, pages 229–240, 2001.

[6] J. Chen, G. Soundararajan, and C. Amza. Autonomic Provisioning of Backend Databases in Dynamic Content Web Servers. In *IEEE International Conference on Autonomic Computing (ICAC)*, pages 231–242, June 2006.

[7] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper. Workload analysis and demand prediction of enterprise data center applications. In *Proceedings of 2007 IEEE International Symposium on Workload Characterization (IISWC'2007)*, Boston, MA, USA, September 2007.

[8] J. Hellerstein, F. Zhang, and P. Shahabuddin. An Approach to Predictive Detection for Service Management. In *Proceedings of the IEEE Intl. Conf. on Systems and Network Management*, 1999.

[9] J. Heo, D. Henriksson, X. Liu, and T. Abdelzaher. Integrating adaptive components: An emerging challenge in performance-adaptive systems and a server farm case-study. In *Proc. of IEEE RTSS, Tuscon, AZ*, December 2007.

[10] D. Irwin, J. Chase, L. Grit, A. Yumerefendi, D. Becker, and K. Yocum. Sharing networked resources with brokered leases. In *Proc. of USENIX Annual Technical Conference*, June 2006.

[11] L. Kleinrock. *Queueing Systems, Volume 2: Computer Applications*. John Wiley and Sons, Inc., 1976.

[12] S. Kumar, V. Talwar, V. Kumar, P. Ranganathan, and K. Schwan. vmanage: Loosely-coupled platform and virtualization management in data centers. In *Proc. of ICAC 2009*, pages 127–136, June 2009.

[13] D. Menasce and F. Ribeiro. In search of invariants for e-business workloads. In *Proceedings of the 2nd ACM conference on Electronic Commerce*, pages 56–65, 2000.

[14] D. A. Menascé, V. A. F. Almeida, R. Fonseca, and M. A. Mendes. A methodology for workload characterization of e-commerce sites. In *EC '99: Proceedings of the 1st ACM conference on Electronic commerce*, pages 119–128, New York, NY, USA, 1999. ACM.

[15] D. Mosberger and T. Jin. httperf - a tool for measuring web server performance. Technical Report HPL-98-61, HP Labs, 1998.

[16] ObjectWeb. the objectweb tpc-w implementation. Website. `http://jmob.objectweb.org/tpcw.html`.

[17] S. Ranjan, J. Rolia, H. Fu, and E. Knightly. Qos-driven server migration for internet data centers. In *Proceedings of IWQoS 2002, Miami Beach, FL*, May 2002.

[18] P. Ruth, X. Jiang, D. Xu, and S. Goasguen. Virtual distributed environments in a shared infrastructure. *IEEE Computer, Special Issue on Virtualization Technologies*, May 2005.

[19] A. B. Sharma, R. Bhagwan, M. Choudhury, L. Golubchik, R. Govindan, and G. M. Voelker. Automatic request categorization in internet services. *SIGMETRICS Perform. Eval. Rev.*, 36(2):16–25, 2008.

[20] C. Stewart, T. Kelly, and A. Zhang. Exploiting nonstationarity for performance prediction. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 31–44, New York, NY, USA, 2007. ACM.

[21] C. Stewart and K. Shen. Performance Modeling and System Management for Multi-component Online Services. In *Proc. USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, May 2005.

[22] TPC. the tpcw benchmark. Website. `http://www.tpc.org/tpcw/`.

[23] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal. Dynamic provisioning for multi-tier internet applications. In *Proceedings of the 2nd IEEE International Conference on Autonomic Computing (ICAC-05), Seattle, WA*, June 2005.

[24] D. Villela, P. Pradhan, and D. Rubenstein. Provisioning Servers in the Application Tier for E-commerce Systems. In *Proceedings of the 12th IWQoS*, June 2004.

[25] Q. Zhang, L. Cherkasova, and E. Smirni. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In *ICAC '07: Proceedings of the Fourth International Conference on Autonomic Computing*, Washington, DC, USA, 2007.