Agile, Dynamic Provisioning of Multi-tier Internet Applications

Bhuvan Urgaonkar, **Prashant Shenoy**, **Abhishek Chandra**†, and **Pawan Goyal**‡

Dept. of Computer Science,
University of Massachusetts,
Amherst, MA 01003
{bhuvan,shenoy}@cs.umass.edu

†Dept. of CSE,
University of Minnesota,
Minneapolis, MN 55455
chandra@cs.umn.edu

‡Veritas Software India Pvt. Ltd.,
Pune, INDIA
pawan.goyal@veritas.com

Abstract

Dynamic capacity provisioning is a useful technique for handling the multi-time-scale variations seen in Internet workloads. In this paper, we propose a novel dynamic provisioning technique for multi-tier Internet applications that employs (i) a flexible queuing model to determine how much resources to allocate to each tier of the application, and (ii) a combination of predictive and reactive methods that determine when to provision these resources, both at large and small time scales. We propose a novel data center architecture based on virtual machine monitors to reduce provisioning overheads. Our experiments on a forty-machine Linux-based hosting platform demonstrate the responsiveness of our technique in handling dynamic workloads. In one scenario where a flash crowd caused the workload of a three-tier application to double, our technique was able to double the application capacity within five minutes, thus maintaining response time targets. Our technique also reduced the overhead of switching servers across applications from several minutes to less than a second, while meeting the performance targets of residual sessions.

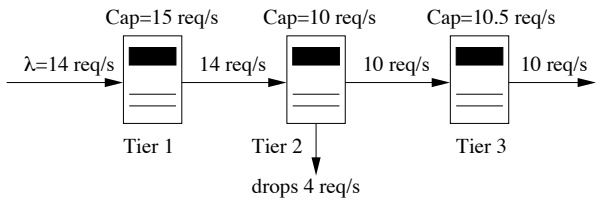
1 Introduction

1.1 Motivation

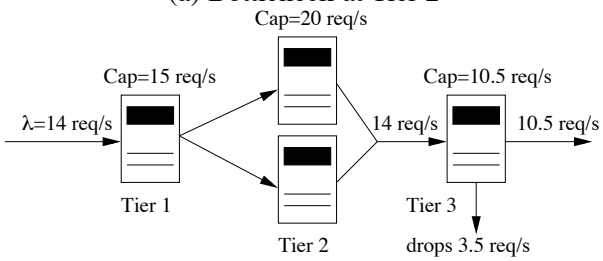
An Internet hosting platform is a server farm that runs distributed applications such as an online retail store or an online brokerage site. Typical Internet applications employ a multi-tier architecture, with each tier providing a certain functionality. Such applications tend to see dynamically varying workloads that contain long-term variations such as time-of-day effects as well as short-term fluctuations due to flash crowds. Predicting the peak workload of an Internet application and capacity provisioning based on these worst case estimates is notoriously difficult. There are numerous documented examples of Internet applications that faced an outage due to an unexpected overload. For instance, the normally well-provisioned Amazon.com site suffered a forty-minute down-time due to an overload during the popular holiday season in November 2000.

Given the difficulties in predicting peak Internet workloads, an application needs to employ a combination of dynamic provisioning and request policing to handle workload variations. Dynamic provisioning enables additional resources—such as servers—to be allocated to an application on-the-fly to handle workload increases, while policing enables the application to temporarily turn away excess requests while additional resources are being provisioned.

In this paper, we focus on dynamic resource provisioning of Internet applications that employ a multi-tier architecture. We argue that (i) provisioning of multi-tier applications raises new challenges not addressed by prior work on provisioning single-tier applications, and (ii) agile, proactive provisioning techniques are necessary to handle both long-term and short-term workload fluctuations seen by Internet applications. To address these issues, we present predictive and reactive provisioning mechanisms as well as a novel data center architecture based on virtual machine monitors.



(a) Bottleneck at Tier 2



(b) Bottleneck shifts to Tier 3

Figure 1: Strawman approaches for provisioning a multi-tier application.

1.2 The Case for A New Provisioning Technique

Dynamic provisioning of resources—allocation and deallocation of servers to replicated applications—has been studied in the context of single-tier applications, of which clustered HTTP servers are the most common example. The notion of *hot spares* and their allocation to cluster-based applications *on-demand* was first proposed in [6]. The Muse project proposed a utility-based approach based on an economic theory for allocation and deallocation of servers to clustered Web servers [3]. A model-based approach for resource provisioning in single-tier Web servers was proposed in [4]. [8] models a multi-tier ecommerce application as a single M/GI/1 server. Whereas multi-tier Internet applications have been studied in the context of SEDA [19, 20], the effort focused on admission control issues to maintain target response times and did not explicitly consider provisioning issues.

It is non-trivial to extend provisioning mechanisms designed for single-tier applications to multi-tier scenarios. To understand why, we consider two strawman approaches that are simple extensions of the above single-tier methods and demonstrate their limitations for multi-tier applications.

Since many single-tier provisioning mechanisms have already been proposed, a straightforward extension is to employ such an approach at each tier of the application. This enables provisioning decisions to be made independently at each tier based on local observations. Thus, our first strawman is to provision additional servers at a tier when the incoming request rate at that tier exceeds the currently provisioned capacity; this can be inferred by monitoring queue lengths, tier-specific response times, or request drop rates. We refer to this approach as *independent per-tier provisioning*.

Example 1: Consider a three-tier Internet application depicted in Figure 1(a). Initially, let us assume that one server each is allocated to the three tiers, and this enables the application to service 15, 10 and 10.5 request/s at each tier (since a user request may impose different demands at different tiers, the provisioned capacity at each tier may be different). Let the incoming request rate be 14 req/s. Given the above capacities, all requests are let in through the first tier, and 4 req/s are dropped at the second tier. Due to these drops, the third tier sees a reduced request rate of 10 req/s and is able to service them all. Thus, the effective goodput is 10 req/s. Since request drops are only seen at the second tier, this tier is perceived to be the bottleneck. The provisioning algorithm at that tier will allocate an additional server, doubling its effective capacity to 20 req/s. At this point, the first two tiers are able to service all incoming requests and the third tier now sees a request rate of 14 req/s (see Figure 1(b)). Since its capacity is only 10.5 req/s, it drops 3.5 req/s. Thus, the bottleneck shifts to the third tier, and the effective goodput only increases from 10 to 10.5 reqs/s.

This simple example demonstrates that increasing the number of servers allocated to the bottleneck tier does not necessarily increase the effective goodput of the application. Instead, it may merely shift the bottleneck to another downstream tier. Although the provisioning mechanism at this downstream tier will subsequently increase its capacity, such shifting bottlenecks may require a number of independent provisioning steps at various tiers before the effective application capacity is actually increased. In the worst case, upto k provisioning steps, one at each tier, may be necessary in a k-tier application. Since allocation of servers to a tier entails overheads of several minutes or more [3], and since Internet workloads may spike suddenly, independent per-tier provisioning may be simply too slow to effectively respond to such workload dynamics.

Our second strawman models the multi-tier application as a black box and allocates additional servers whenever the observed response time exceeds a threshold.

Example 2: Consider the same three-tier application from Example 1 with tier-specific capacities of 15, 20 and 10.5 req/s as depicted in Figure 1(b). We ignore admission control issues in this example. Since the incoming request rate is 14 req/s, the first two tiers are able to serve all requests, while the third tier saturates, causing request queues to build up at this tier. This queue build up increases the end-to-end response time of the application beyond the threshold. Thus, like in the single-tier case, a black box approach can successfully detect when additional servers need to be provisioned for the multi-tier application.

However, determining *how many* servers to provision and *where* is far more complex for multi-tier applications. First, since the application is treated as a black box, the provisioning mechanism can only detect an increase in end-to-end response times but can not determine which tier is responsible for this increase. Second, for single-tier applications, an application model is used to determine how many servers are necessary to service all incoming requests with a certain response time threshold [4]. Extending such models to multi-tier applications is non-trivial, since each tier has different characteristics. In a typical ecommerce application, for instance, this implies *collectively* modeling the effects of HTTP servers, Java application servers and database servers—a complex task. Third, not all tiers of the application may be replicable. For instance, the database tier is typically difficult to replicate on-the-fly. In the above example, if the third tier is a non-replicable database, the black box approach, which has no knowledge of individual tiers, will incorrectly signal the need to provision additional servers, when the correct action is to trigger request policing and let no more than 10.5 req/s into the "black box".

The above example demonstrates that due to the very nature of multi-tier applications, it is not possible to treat them as a black box for provisioning purposes. Knowledge of the number of tiers, their current capacities, and constraints on the degree of replication at each tier is essential for making proper provisioning decisions.

Both examples expose the limitations of using variants of single-tier provisioning methods for multi-tier applications. This paper presents a multi-tier provisioning technique that overcome these limitations.

1.3 Research Contributions

This paper addresses the problem of dynamically provisioning capacity to a multi-tier application so that it can service its peak workload demand while meeting contracted response time guarantees. The provisioning technique proposed in this paper is tier-aware, agile, and is able to take any tier-specific replication constraints into account. Our work has led to the following research contributions.

Predictive and Reactive Provisioning: Our provisioning technique employs two methods that operate at two different time scales—predictive provisioning that allocates capacity at the time-scale of hours or days, and reactive provisioning that operates at time scales of minutes to respond to flash crowds or deviations from expected long-term behavior. The combination of predictive and reactive provisioning is a novel approach for dealing with the multi-time-scale variations in Internet workloads.

Analytical modeling and incorporating tails of workload distributions: We present a flexible analytical model based on queuing theory to capture the behavior of applications with an *arbitrary* number of tiers. Our model determines the number of servers to be allocated to each tier based on the estimated workload. A novel aspect of our model-based provisioning is that it is based on the *tail* of the workload distribution—since capacity is usually engineered for the worst-case load, we use tails of probability distributions to estimate peak demand.

Virtual Machine based provisioning: While virtual machines (VMs) have long been used to share a physical server among multiple applications, we use VMs for a novel purpose, namely agile switching of servers among applications. Traditional methods to switch a server from an underloaded to an overloaded application have entailed latencies of several minutes or more, due to software installation and configuration overheads. We run applications inside virtual machines and use the resource partitioning features of the underlying virtual machine monitor to quickly "switch" a server from one application to another. This enables our system to be extremely agile to load spikes with reaction times of tens of milliseconds.

Handling session-based workloads: Modern Internet workloads are predominantly session based. Consequently, our techniques are inherently designed to handle session-based workloads—they can account for multiple requests that comprise a session and the stateful nature of session-based Internet applications.

Implementation and experimentation: We implement our techniques on a forty-machine Linux-based hosting platform and use our prototype to conduct a detailed experimental evaluation using two open-source multi-tier applications. Our results show: (i) our model effectively captures key characteristics of multi-tier applications and overcomes the shortcomings inherent in existing provisioning techniques based on single-tier models, (ii) the combination of predictive and reactive mechanisms allows us to deal with predictable workload variations as well as unexpected spikes (during a flash crowd, our data center could double the capacity of a three-tier application within 5 min), and (iii) the use of virtual machines can reduce the overhead of switching servers from several tens of minutes to less than a second.

The remainder of this paper is structured as follows. Section 2 presents an overview of the proposed system. Sections 3-6 present our provisioning algorithms. We present our prototype implementation in 7, our experimental evaluation in Section 8, and conclusions in Section 9.

2 System Overview

This section presents an overview of Internet applications and the hosting platform assumed in our work.

2.1 Multi-tier Internet Applications

Modern Internet applications are designed using multiple tiers. A multi-tier architecture provides a flexible, modular approach for designing such applications. Each 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 2). For example, an online bookstore can be designed using three tiers—a front-end Web server responsible for HTTP processing, a middle-tier Java application server that implements the application logic, and a backend database that stores catalogs and user orders.

The various tiers of an application are assumed to be distributed across different servers. Depending on the desired capacity, a tier may also be clustered. In an online bookstore, for example, the front-end tier can be a clustered Apache server that runs on multiple machines. 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. Not all tiers may be replicable. For instance, if the backend tier of the bookstore employs a database with *shared-nothing* architecture, it cannot be replicated on-demand. Database servers with a shared-everything architecture [11], in contrast, can be clustered and replicated on-demand, but with certain constraints. We assume that each tier specifies its degree of replication, which is the limit on the maximum number of servers that can be allocated to it.[1]

Each clustered tier is also assumed to employ a load balancing element that is responsible for distributing requests to servers in that tier [12]. The workload of an Internet application is assumed to be session-based, where a session consists of a succession of requests issued by a client with think times in between. If a session is stateful, successive requests will need to be serviced by the same server at each tier, and the load balancing element will need account for this server state when redirecting requests.

[1]The degree of replication of a tier can vary from one to infinity, depending on whether the tier is partially, infinitely, or not replicable.

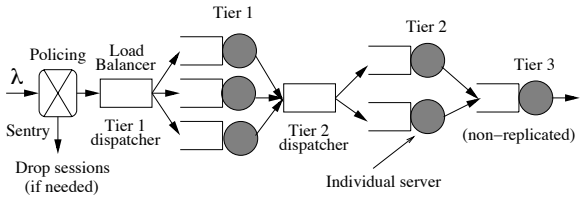


Figure 2: Architecture of a 3-tier Internet Application. In this example, tiers 1 and 2 are clustered, while tier 3 is not.

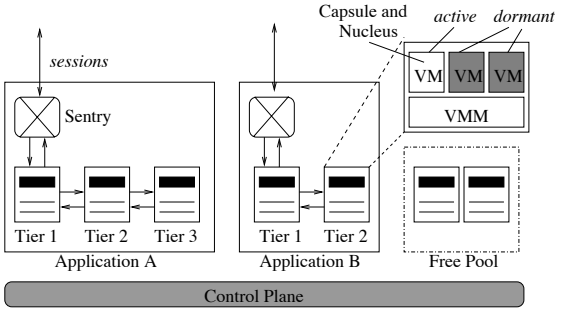


Figure 3: Hosting Platform Architecture.

Every application also runs a special component called a sentry. The sentry polices incoming sessions to an application's server pool—incoming sessions are subjected to admission control at the sentry to ensure that the contracted performance guarantees are met; excess sessions are turned away during overloads (see Figure 2). Observe that, unlike systems that use per-tier admission control [19], we assume a policer that makes a one-time admission decision when a session arrives. Once a session has been admitted, none of its requests can be dropped at any intermediate tier. Thus, sufficient capacity needs to be provisioned at various tiers to service all admitted sessions. Such a one-time policer avoids resource wastage resulting from partially serviced requests that may be dropped at later tiers.

Finally, we assume that each application desires a performance bound from the hosting platform that is specified in the form of a service-level agreement (SLA). Our work assumes that the SLA is specified either in terms of the average response time or a suitable high percentile of the response time distribution (e.g., a SLA may specify that 95% of the requsts should incur an end-to-end response time of no more than 1 second).

2.2 Hosting Platform Architecture

Our hosting platform is a data center that consists of a cluster of commodity servers interconnected by gigabit Ethernet. One or more high bandwidth links connect this cluster to the Internet. Each server in the hosting platform can take on one of the following roles: run an application component, run the control plane, or be part of the free pool (see Figure 3). The free pool contains all unallocated servers.

Servers Hosting Application Components: The hosting platform runs multiple third-party applications concurrently in return for revenues [3, 16, 17]. This work assumes a dedicated hosting model, where each application runs on a subset of the servers and a server is allocated to *at most* one application at any given time (except in special circumstances that we explain later).[2] The dedicated model is useful for running large clustered applications such as online mail [14], retail and brokerage sites, where server sharing is infeasible due to the client workload—the server pool is partitioned among applications running on the platform.

[2]A dedicated hosting model is different from shared hosting [17] where the number of application exceeds the number of servers, and each server may run multiple applications concurrently.

The component of an application that runs on a server is referred to as a *capsule*. Each capsule runs inside a virtual machine and each server runs a virtual machine monitor that executes this virtual machine. Depending on whether the capsule is replicable or not, the server may get classified as an *Elf* or an *Ent*. *Elf* servers runs replicable caspules, while *Ents* run non-replicable components of an application.[3] Unlike Ents, an Elf can be reassigned from one application to another. Multiple VMs and their associated capsules may reside on an Elf, although only one of these VMs can be active at any given time, as per the dedicated hosting model. The remaining VMs are dormant and are assigned minimal server resources. Dormant VMs enable fast server switching in the presence of dynamic workloads as explained in Section 6. Each VM also runs a *nucleus*—a software component that performs online measurements of the capsule workload, its performance and resource usage; these statistics are periodically conveyed to the control plane.

Control Plane: The control plane is responsible for dynamic provisioning of servers to individual applications. It tracks the resource usage on servers, as reported by the nuclei, and determines the number of servers to be allocated to each application.

3 Provisioning Algorithm Overview

The goal of our provisioning algorithm is to allocate sufficient capacity to the tiers of an application so that its SLA can be met even in the presence of the peak workload. At the heart of any provisioning algorithm lie two issues: *how much* to provision and *when*? We provide an overview of our provisioning algorithm from this perspective.

How much to provision: To address the issue of how many servers to allocate to each tier and each application, we construct an analytical model of an Internet application. Our model takes as input the incoming request rate and service demand of an individual request, and computes the number of servers needed at each tier to handle the aggregate demand.

We model a multi-tier application as a network of queues where each queue represents an application tier (more precisely, a server at an application tier), and the queues from a tier feeds into the next tier. We model a server at a tier using as a G/G/1 system, since it is sufficiently general to capture arbitrary arrival distributions and service time distributions.

By using this building block, which we describe in Section 4, we determine the number of servers necessary at each tier to handle a peak session arrival rate of λ and provision resources accordingly. Our approach overcomes the drawbacks of independent per-tier provisioning and the black box approaches: (1) While the capacity needed at each tier is determined separately using our queuing model, the desired capacities are allocated to the various tiers all at once. This ensures that each provisioning decision immediately results in an increase in effective capacity of the application. (2) The use of a G/G/1 building block for a server at each tier enables us to break down the complex task of modeling an arbitrary multi-tier application into more manageable units. Our approach retains the ability to model each tier separately, while being able to reason about the needs of the application as a whole.

When to Provision: The decision of when to provision depends on the dynamics of Internet workloads. Internet workloads exhibit long-term variations such as time-of-day or seasonal effects as well as short-term fluctuations such as flash crowds. While long-term variations can be predicted ahead of time by observing past variations, short-term fluctuations are less predictable, or in some cases, not predictable. Our techniques employ two different methods to handle variations observed at different time scales. We use predictive provisioning to estimate the workload for the next few hours and provision for it accordingly. Reactive provisioning is used to correct errors in the long-term predictions or to react to unanticipated flash crowds. Whereas predictive provisioning attempts to "stay ahead" of the anticipated workload fluctuations, reactive provisioning enables the hosting platform to be agile to deviations from the expected workload.

The following sections present our queuing model, and the predictive and reactive provisioning methods.

[3]Elves are a swift and athletic race in J.R.R. Tolkien's *The Lord of the Rings* as opposed to the bulky, tree-like Ents.

4 How Much to Provision: Modeling Multi-tier Applications

To determine how many servers to provision for an application, we present an analytical model of a multi-tier application. Consider an application that consists of k tiers, denoted by $T_1, T_2, \ldots T_k$. Let the desired end-to-end response time for the application be R; this value is specified by the application's contracted SLA. Assume that the end-to-end response time is broken down into per-tier response times[4], denoted by $d_1, d_2, \ldots, d_k$, such that $\sum d_i = R$. Let the incoming session rate be λ. Since capacity is typically provisioned based on the worst case demand, we assume that λ is some high percentile of the arrival rate distribution—an estimate of the peak session rate that will be seen by the application.

Given the (peak) session rate and per-tier response times, our objective is to determine how many servers to allocate such that each tier can service all incoming requests with a mean response time of d_i.

Our model is based on a network of queues. Each server allocated to the application is represented by a queue (see Fig 2). Queues (servers) representing one tier feed into the those representing the next tier. The first step in solving our model is to determine the capacity of an individual server in terms of the request rate it can handle. Given the capacity of a server, the next step computes the number of servers required at a tier to service the peak session rate. We model each server as a G/G/1 queuing system. Since a G/G/1 system can handle an arbitrary arrival distribution and arbitrary service times, it enables us to capture the behavior of a various tiers such as HTTP, J2EE, and database servers.

The behavior of a G/G/1 system can be captured using the following queuing theory result [9]:

$$\lambda_i \geq \left[s_i + \frac{\sigma_a^2 + \sigma_b^2}{2 \cdot (d_i - s_i)} \right]^{-1} \tag{1}$$

where d_i is the mean response time for tier i, s_i is the average service time for a request a that tier, and λ_i is the request arrival rate to tier i. σ_a^2 and σ_b^2 are the variance of inter-arrival time and the variance of service time, respectively.

Observe that d_i is known, while the per-tier service time s_i as well as the variance of inter-arrival and service times σ_a^2 and σ_b^2 can be monitored online in our system. By substituting these values into Equation 1, a lower bound on request rate λ_i that can serviced by a single server can be obtained.

Given an average session think-time of Z, a session issues requests at a rate of $\frac{1}{Z}$. Using Little's Law [9], we can translate the *session* arrival rate of λ to a *request* arrival rate of $\frac{\lambda\tau}{Z}$, where τ is the average session duration. Therefore, once the capacity of a single server λ_i has been computed, the number of servers η_i needed at tier i to service a peak request rate of $\frac{\lambda\tau}{Z}$ is simply computed as

$$\eta_i = \left\lceil \frac{\beta_i \lambda \tau}{\lambda_i Z} \right\rceil \tag{2}$$

where β_i is a tier-specific constant. The quantities Z and τ are estimated using online measurements.

Observe that a single incoming request might trigger more than one request (unit of work) at intermediate tiers. For instance, a single search request at an online superstore might trigger multiple queries at the backend database, one in the book catalog, one in the music catalog and so on. Consequently, our model assumes that $\frac{\lambda\tau}{Z}$ incoming requests impose an aggregate demand of $\beta_1 \frac{\lambda\tau}{Z}$ requests at tier 1, $\beta_2 \frac{\lambda\tau}{Z}$ requests at tier 2 and so on. The parameters $\beta_1, \ldots \beta_k$ are derived using online measurements. The value of β_i may be greater that one than if a request triggers multiple units of work at tier i or it may be less than one if caching at prior tiers reduces the demand at this tier.

Observe that our model can handle applications with an arbitrary number of tiers, since the complex task of modeling a multi-tier application is reduced to modeling an individual server at each tier. Equation 2 assumes that servers are homogeneous and that servers in each tier are load-balanced. Both assumptions can be relaxed; these extensions are omitted due to space constraints.

The output of the model is the number of servers $\eta_1, \ldots \eta_k$ needed at the k tiers to handle a peak demand of λ. We then increase the capacity of all tiers to these values in a single step, resulting in an immediate increase in effective

[4] Offline profiling can be used to break down the end-to-end response time into tier-specific response times.

capacity. In the event η_i exceeds the degree of replication M_i of a tier the actual allocation is reduced to this limit. Thus, each tier is allocated no more than $min(\eta_i, M_i)$ servers. To ensure that the SLA is not violated when the allocation is reduced to M_i, the excess requests must be turned away at the sentry.

5 When to Provision?

In this section, we present two methods—predictive and reactive—to provision resources over long and short time-scales, respectively.

5.1 Predictive Provisioning for the Long Term

The goal of predictive provisioning is to provision resources over time scales of hours and days. The technique uses a workload predictor to predict the peak demand over the next several hours or a day and then uses the model presented in Section 4 to determine the number of servers that are needed to meet this peak demand. Predictive provisioning is motivated by long-term variations such as time-of-day or seasonal effects exhibited by Internet workloads [7]. For instance, the workload seen by an Internet application typically peaks around noon every day and is minimum in the middle of the night. Similarly, the workload seen by online retail Web sites is higher during the holiday shopping months of November and December than other months of the year. These cyclic patterns tend to repeat and can be predicted ahead of time by observing past variations. By employing a workload predictor that can predict these variations, our predictive provisioning technique can allocate servers to an application well ahead of the expected workload peak. This ensures that application performance does not suffer even under the peak demand.

The key to predictive provisioning is the workload predictor. In this section, we present a workload predictor that estimates the tail of the arrival rate distribution (i.e., the peak demand) for the next few hours. Other statistical workload predictive techniques proposed in the literature can also be used with our predictive provisioning technique [7, 13].

Our workload predictor is based on a technique proposed in [13] and uses past observations of the workload to predict peak demand that will be seen over a period of T hours. For simplicity of exposition, assume that $T = 1$ hour. In that case, the predictor estimates the peak demand that will be seen over the next one hour, at the beginning of each hour. To do so, it maintains a history of the session arrival rate seen during each hour of the day over the past several days. A histogram is then generated for each hour using observations for that hour from the past several days (see Figure 4). Each histogram yields a probability distribution of the arrival rate for that hour. The peak workload for a particular hour is estimated as a high percentile of the arrival rate distribution for that hour (see Figure 4). Thus, by using the tail of the arrival rate distribution to predict peak demand, the predictive provisioning technique can allocate sufficient capacity to handle the worst-case load, should it arrive. Further, monitoring the demand for each hour of the day enables the predictor to capture time-of-day effects.

In addition to using observations from prior days, the workload seen in the past few hours of the current day can be used to further improve prediction accuracy. Suppose that $\lambda_{pred}(t)$ denotes the predicted arrival rate during a particular hour denoted by t. Further let $\lambda_{obs}(t)$ denote the actual arrival rate seen during this hour. The prediction error is simply $\lambda_{obs}(t) - \lambda_{pred}(t)$. In the event of a consistently *positive* prediction error over the past few hours, indicating that the predictor is consistently underestimating peak demand, the predicted value for the next hour is corrected using the observed error:

$$\lambda_{pred}(t) = \lambda_{pred}(t) + \sum_{i=t-h}^{t-1} \frac{max(0, \lambda_{obs}(i) - \lambda_{pred}(i))}{h}$$

where the second expression denotes the mean prediction error over the past h hours. We only consider positive errors in order to correct underestimates of the predicted peak demand—negative errors indicate that the observed workload is *less* than the peak demand, which only means that the worst-case workload did not arrive in that hour and is not necessarily a prediction error.

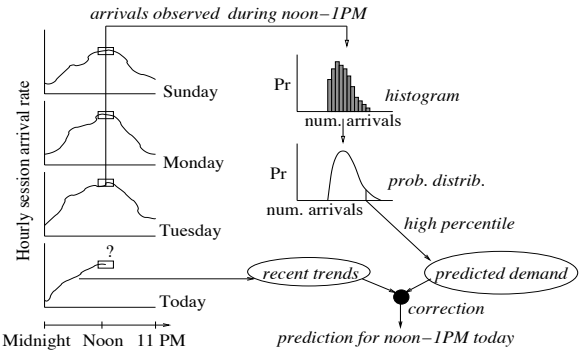


Figure 4: The workload prediction algorithm.

Using the predicted peak arrival rate for each application, the predictive provisioning technique uses the model to determine the number of servers that should be allocated to each tier of an application. An increase in allocation must be met by borrowing servers from the free pool or underloaded applications—underloaded applications are those whose new allocations are less than their current allocations. If the total number of required servers is less than the servers available in the free pool and those released by underloaded applications, then a utility-based approach [3] can be used to arbitrate the allocation of available servers to needy applications—servers are allocated to applications that benefit most from it as defined by their utility functions.

5.2 Reactive Provisioning: Handling Prediction Errors and Flash Crowds

The workload predictor outlined in the previous section is not perfect—it may incur prediction errors if the workload on a given day deviates from its behavior on previous days. Further, sudden load spikes or flash crowds are inherently unpredictable phenomena. Finally, errors in the online measurements of the model parameters can translate into errors in the allocations computed by the model. Reactive provisioning is used to swiftly react to such unforeseen events. Reactive provisioning operates on short time scales—on the order of minutes—checking for workload anomalies. If any such anomalies are detected, then it allocates additional capacity to various tiers to handle the workload increase.

Reactive provisioning is invoked once every few minutes. It can also be invoked on-demand by the application sentry if the observed request drop rate increases beyond a threshold. In either case, it compares the currently observed session arrival rate $\lambda_{obs}(t)$ over the past few minutes to the predicted rate $\lambda_{pred}(t)$. If the two differ by more than a threshold, corrective action is necessary. Specifically if $\frac{\lambda_{obs}(t)}{\lambda_{pred}(t)} > \tau_1$ or drop rate $> \tau_2$, where τ_1 and τ_2 are application-defined thresholds, then it computes a new allocation of servers. This can be achieved in one of two ways. One approach is to use the observed arrival rate $\lambda_{obs}(t)$ in Equation 2 of the model to compute a new allocation of servers for the various tiers. The second approach is to increase the allocation of all tiers that are at or near saturation by a constant amount (e.g., 10%). The new allocation needs to ensure that the bottleneck does not shift to another downstream tier; the capacity of any such tiers may also need to be increased proportionately. The advantage of using the model to compute the new allocation is that it yields the new capacity in a single step, as opposed to the latter approach that increases capacity by a fixed amount. The advantage of the latter approach is that it is independent of the model and can handle any errors in the measurements used to parameterize the model. In either case, the effective capacity of the application is raised to handle the increased workload.

The additional servers are borrowed from the free pool if available. If the free pool is empty or has insufficient servers, then these servers need to be borrowed from other underloaded applications running on the hosting platform. An application is said to be underloaded if its observed workload is significantly lower than its provisioned capacity: if $\frac{\lambda_{obs}(t)}{\lambda_{pred}(t)} < \tau_{low}$, where τ_{low} is a low watermark threshold.

Since a single invocation of reactive provisioning may be insufficient to bring sufficient capacity online during a

large load spike, repeated invocations may be necessary in quick succession to handle the workload increase.

Together, predictive and reactive provisioning can handle long-term predictable workload variations as well as short term fluctuations that are less predictable. Predictive provisioning allocates capacity ahead of time in anticipation of a certain peak workload, while reactive provisioning takes corrective action *after* an anomalous workload increase has been observed. Put another way, predictive provisioning attempts to stay ahead of the workload fluctuations, while reactive provisioning follows workload fluctuations correcting for errors.

5.3 Request Policing

The predictor and reactor convey the peak session arrival rate for which they have allocated capacity to the application's sentry. This is done every time the allocation is changed. The sentry then ensures that the admission rate does not exceed this threshold—excess sessions are dropped at the sentry.

6 Agile Server Switching using VMMs

A Virtual Machine Monitor (VMM) is a software layer that virtualizes the resources of a physical server and supports the execution of multiple virtual machines (VMs) [2]. Each VM runs a separate operating system and an application capsule within it. The VMM enables servers resources, such as the CPU, memory, disk and network bandwidth, to be partitioned among the resident virtual machines.

Traditionally VMMs have been employed in shared hosting environments to run multiple applications and their VMs on a single server; the VM provides isolation across applications while the VMM supports flexible partitioning of server resources across applications. In dedicated hosting, no more than one application can be active on a given physical server, and as a result, sharing of individual server resources across applications is moot in such environments. Instead, we employ VMMs for a novel purpose—fast server switching.

Traditionally, switching a server from one application to another for purposes of dynamic provisioning has entailed overheads of several minutes or more. Doing so involves some or all of the following steps: (i) wait for residual sessions of the current application to terminate, (ii) terminate the current application, (iii) scrub and reformat the disk to wipe out sensitive data, (iv) reinstall the OS, (v) install and configure the new application. Our hosting platform runs a VMM on each physical server. Doing so enables it to eliminate many of these steps and drastically reduces switching time.

We assume that each Elf server runs multiple virtual machines and capsules of different applications within it. Only one capsule and its virtual machine is active at any time—this is the capsule to which the server is currently allocated. Other virtual machines are dormant—they are allocated minimal server resources by the underlying VMM and most server resources are allocated to the active VM. If the server belongs to the free pool, all of its resident VMs are dormant.

In such a scenario, switching an Elf server from one application to another implies deactivating a VM by reducing its resource allocation to ϵ, and reactivating a dormant VM by increasing its allocation to 100-ϵ% of the server resources.[5] This only involves adjusting the allocations in the underlying VMM and incurs overheads in the order of tens of milliseconds. Thus, in theory, our hosting platform can switch a server from one application to another in a few milliseconds. In practice, however, we need to consider the residual state of the application before it can be made dormant.

To do so, we assume that once the predictor or the reactor decide to reassign a server from an underloaded to an overloaded application, they notify the load balancing element of the underloaded application tier. The load balancing element stops forwarding new sessions to this server. However, the server retains state of existing sessions and new requests may arrive for those sessions until they terminate. Consequently, the underloaded application tier will continue to use some server resources and the amount of resources required will diminish over time as existing sessions terminate. As a result, the allocation of the currently active VM can not be instantaneously ramped down;

[5] ϵ is a small value such that the VM consumes negligible server resources and its capsule is idle and swapped out to disk.

instead the allocation needs to be reduced gradually, while increasing the allocation of the VM belonging to the overloaded application. Two strategies for ramping down the allocation of the current VM are possible.

- *Fixed rate ramp down:* In this approach, the resource allocation of the underloaded VM is reduced by a fixed amount δ every t time units until it reduces to ϵ; the allocation of the new VM is increased correspondingly. The advantage of this approach is that it switches the server from one application to another in a fixed amount of time, namely t/δ. The limitation is that long-lived residual sessions will be forced to terminate, or their performance guarantees will be violated if the allocation decreases beyond that necessary to service them.

- *Measurement-based ramp down:* In this approach, the actual resource usage of the underloaded VM is monitored online. As the resource usage decreases with terminating sessions, the underlying allocation in the VMM is also reduced. The approach requires monitoring of the CPU, memory, network and disk usage so that the allocation can match the falling usage. The advantage of this approach is that the ramp-down is more conservative and less likely to violate performance guarantees of existing sessions. The drawback is that long-lived sessions may continue to use server resources, which increases the server switching time.

In either case, use of VMMs enables our hosting platform to reduce system switching overheads to a negligible value. The switching time is solely dominated by application idiosyncrasies. If the application has short-lived sessions or the application tier is stateless, the switching overhead is small. Even when sessions are long-lived, the overloaded application immediately gets some resources on the server, which increases its effective capacity; more resources become available as the current VM ramps down.

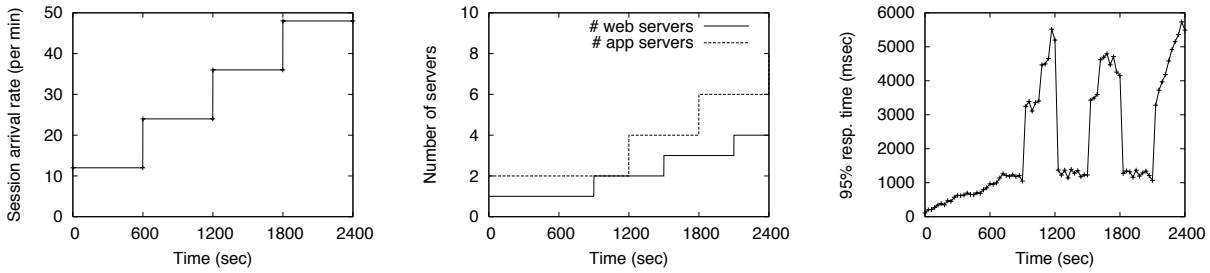
As a final detail, observe that we have assumed that sufficient dormant VMs are always available for various tiers of an overloaded application to arbitrarily increase its capacity. The hosting platform needs to ensure that there is always a pre-spawned pool of dormant VMs for each application in the system. As dormant VMs of an application are activated during an overload, and the number of dormant VMs falls below a low watermark, additional dormant VMs need to be spawned on other Elf servers, so that there is always a ready pool of VMs that can be tapped.

7 Implementation Considerations

We implemented a prototype data center on a cluster of 40 Pentium servers connected via a 1Gbps ethernet switch and running Linux 2.4.20. Each machine in the cluster ran one of the following entities: (1) an application capsule (and its nucleus) or load balancer, (2) the control plane, (3) a sentry, (4) a workload generator for an application. The applications used in our evaluation (described in detail in Section 8.1) had two replicable tiers—front tier based on the Apache Web server and a middle tier based on Java servlets hosted on the Tomcat servlets container. The third tier was a non-replicable Mysql database server.

Virtual Machine Monitor: We use Xen 1.2 [2] as the virtual machine monitor in our prototype. The Xen VMM has a special virtual machine called domain0 (virtual machines are called domains in the Xen terminology) that gets created as soon as Xen boots and remains throughout the VMM's existence. Xen provides a management interface that can be manipulated from domain0 to create new domains, control their CPU, network and memory resource allocations, allocate IP addresses, grant access to disk partitions, and suspend/resume domains to files, etc. The management interface is implemented as a set of library functions (implemented in C) for which there are Python language bindings. We use a subset of this interface—`xc_dom_create.py` and `xc_dom_control.py` provide ways to start a new domain or stop an existing one; the control plane implements a script that remotely logs on to domain0 and invokes these scripts. The control plane also implements scripts that can remotely log onto any existing domain to start a capsule and its nucleus or stop them. `xc_dom_control.py` provides an option that can be used to set the CPU share of an existing domain. The control plane uses this feature for VM ramp up and ramp down.

Nucleus: The nucleus was implemented as a user-space daemon that periodically (once every 15 min in our prototype) extracts information about tier-specific requests needed by the provisioning algorithms and conveys it to the control plane. Our nuclei use a combination of (i) online measurements of resource usages and request performance,



(a) Session arrival rate (b) Number of servers (c) Response time

Figure 5: Rubbos: Independent per-tier provisioning

(ii) real-time processing of logs provided by the application software components, and (iii) offline measurements to determine various quantities needed by the control plane. We made simple modifications to Apache and Tomcat to record the average service time s_i of a request at these tiers. For Mysql, s_i was determined using offline profiling [17]. The variance of service time, σ_b^2, was determined from observations of individual service times. We configured Apache and Tomcat (by turning on the appropriate options in their configuration files) to have them record the arrival and residence times of individual requests into their logs. The logs were written to named pipes and processed in real-time by the nuclei to determine, σ_a^2, the variance of the request inter-arrival time. The parameter β_i for tier i was estimated by the control plane as the ratio of the number of requests reported by the nuclei at that tier and the number of requests admitted by the sentry during the last period. Finally, the nuclei used the sysstat package [15] for online measurements of resource usages of capsules that is used by the reactive provisioning and by the measurement-based strategy for ramping down the allocation of a VM.

Sentry and Load Balancer: We used *Kernel TCP Virtual Server* (ktcpvs) version 0.0.14 [10] to implement the policing mechanisms described in Section 5.3. ktcpvs is an open-source, Layer-7 request dispatcher implemented as a Linux module. A round-robin load balancer implemented in ktcpvs was used for Apache. Load balancing for the Tomcat tier was performed by *mod_jk*, an Apache module that implements a variant of round robin request distribution while taking into account session affinity. The sentry keeps record of arrival and finish times of admitted sessions as well as each request within a session. These observations are used to estimate the average session duration τ and the average think time Z.

Control Plane: The control plane is implemented as a daemon running on a dedicated machine. It implements the predictive and reactive provisioning techniques described in Section 5. The control plane invokes the predictive provisioning algorithm periodically and conveys the new server allocations or deallocations to the affected sentries and load balancers. It communicates with the concerned virtual machine monitors to start or stop capsules and nuclei. Reactive provisioning is invoked by the sentries once every 5 min.

8 Experimental Evaluation

In this section we present the experimental setup followed by the results of our experimental evaluation.

8.1 Experimental Setup

The control plane was run on a dual-processor 450MHz machine with 1GB RAM. Elf and Ent servers had 2.8GHz processors and 512MB RAM. The sentries were run on dual-processor 1GHz machines with 1GB RAM. Finally, the workload generators were run on uniprocessor machines with 1GHz processors. Elves and Ents ran the Xen 1.2 VMM with Linux; all other machines ran Linux 2.4.20. All machines were interconnected by gigabit Ethernet.

We used two open-source multi-tier applications in our experimental study. *Rubis* implements the core functionality of an eBay like auction site: selling, browsing and bidding. It implements three types of user sessions, has nine tables

in the database and defines 26 interactions that can be accessed from the clients' Web browsers. *Rubbos* is a bulletin-board application modeled after an online news forum like Slashdot. Users have two different levels of access: regular user and moderator. The main tables in the database are the users, stories, comments, and submissions tables. Rubbos provides 24 Web interactions. Both applications were developed by the DynaServer group at Rice University [5]. Each application contains a Java-based client that generates a session-oriented workload. We modified these clients to generate workloads and take measurements needed by our experiments. Rubis and Rubbos sessions had an average duration of 15min and 5min, respectively. For both applications, the average think time was 5sec.

We used 3-tier versions of these applications. The front tier was based on Apache 2.0.48 Web server. The middle tier was based on Java servlets that implement the application logic. We employed Tomcat 4.1.29 as the servlets container. Finally, the database tier was based on the Mysql 4.0.18 database.

Both applications are assumed to require an SLA where the 95^{th} percentile of the response time is no greater than 2 seconds. We use a simple heuristic to translate this SLA into an equivalent SLA specified using the average response time—since the model in Section 4 uses mean response times, such a translation is necessary. We use application profiling [17] to determine a distribution whose 95^{th} percentile is 2 seconds and use the mean of that distribution for the new SLA. The per-tier average delay targets d_1, d_2 and d_3 were then set to be 10, 50 and 40% of the mean response time, for Apache, Tomcat and Mysql respectively.

8.2 Effectiveness of Multi-tier Model

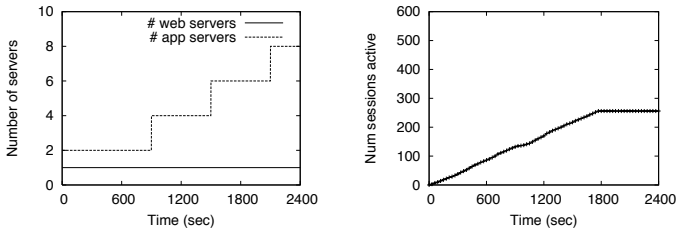
This section demonstrates the effectiveness of our multi-tier provisioning technique over variants of single-tier methods.

8.2.1 Independent per-tier provisioning

Our first experiment uses the Rubbos application. We use the first strawman described in Example 1 of Section 1 for provisioning Rubbos. Here, each tier employs its own provisioning technique. Rubbos was subjected to a workload that increases in steps, once every ten minutes (see Fig. 5(a)). The first workload increase occurs at t=600sec and saturates the tier-1 Web server. This triggers the provisioning technique, and an additional server is allocated at t=900sec (see Figure 5(b)). At this point, the two tier-1 servers are able to service all incoming requests, causing the bottleneck to shift to the Tomcat tier. The Elf running Tomcat saturates, which triggers provisioning at tier 2. An additional server is allocated to tier 2 at t=1200sec (see Fig. 5(b)). The second workload increase occurs at t=1200sec and the above cycle repeats. As shown in Figure 5(c), since multiple provisioning steps are needed to yield an effective increase in capacity, the application SLA is violated during this period.

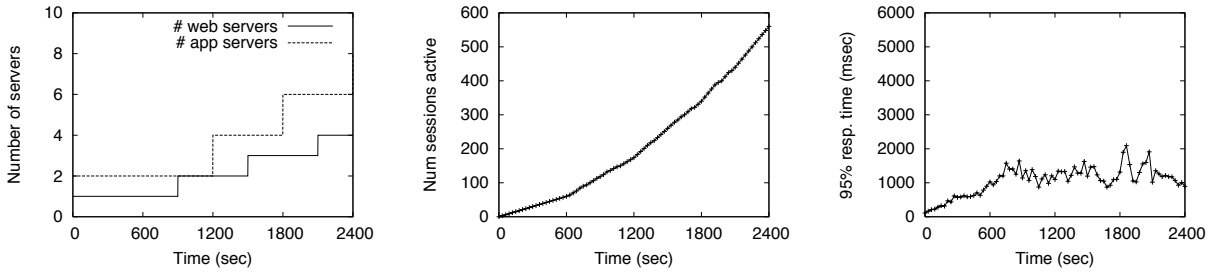
A second strawman is to employ dynamic provisioning only at the most compute-intensive tier of the application, since it is the most common bottleneck [18]. In Rubbos, the Tomcat tier is the most compute intensive of the three tiers and we only subject this tier to dynamic provisioning. The Apache and Tomcat tiers were initially assigned 1 and 2 servers respectively. The capacity of a Tomcat server was determined to be 40 simultaneous sessions using our model, while Apache was configured with a connection limit of 256 sessions. As shown in Figure 6(a), every time the current capacity of the Tomcat tier is saturated by the increasing workload, two additional servers are allocated. The number of servers at tier-2 increases from 2 to 8 over a period of time. At t=1800sec, the session arrival rate increases beyond the capacity of the first tier, causing the Apache server to reach its connection limit of 256. Subsequently, even though plenty of capacity was available at the Tomcat tier, newly arriving sessions are turned away due to the connection bottleneck at Apache and the throughput reaches a plateau (see Figure 6(b)). Thus, focusing only on the the commonly bottlenecked tier is not adequate, since the bottleneck will eventually shift to other tiers.

Next, we repeat this experiment with our multi-tier provisioning technique. Since our technique is aware of the demands at each tier and can take idiosyncrasies such as connection limits into account, as shown in Figure 7(a), it is able to scale the capacity of both the Web and the Tomcat tiers with increasing workloads. Consequently, as shown in Figure 7(b), the application throughput continues to increase with the increasing workload. Figure 7(c) shows that the SLA is maintained throughout the experiment.



(a) Number of servers (b) Num active sessions

Figure 6: Rubbos: Provision only the Tomcat tier



(a) Number of servers (b) Num active sessions (c) Response time

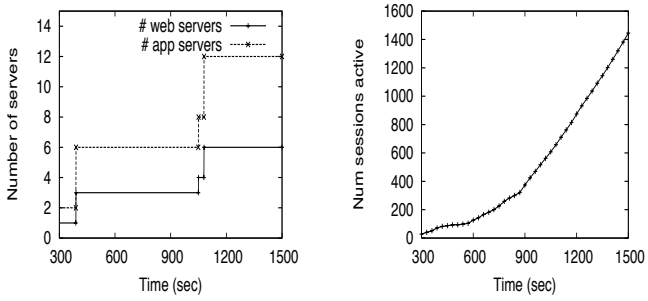
Figure 7: Rubbos: Model-based multi-tier provisioning

Result: Existing single-tier methods are inadequate for provisioning resources for multi-tier applications as they may fail to capture multiple bottlenecks. Our technique anticipates shifting bottlenecks due to capacity addition at a tier and increases capacity at all needy tiers. Further, it can identify different bottleneck resources at different tiers, e.g. CPU at the Tomcat tier and Apache connections at the Web tier.

8.2.2 The Black box Approach

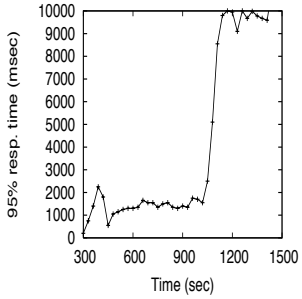
We subjected the Rubis application to a workload that increased in steps, as shown in Figure 9(a). First, we use the black box provisioning approach described in Example 2 of Section 1. The provisioning technique monitors the per-request response times over 30s intervals and signals a capacity increase if the 95^{th} percentile response time exceeds 2sec. Since the black box technique is unaware of the individual tiers, we assume that two Tomcat servers and one Apache server are added to the application every time a capacity increase is signaled. As shown in Figure 8(a) and (c), the provisioned capacity keeps increasing with increasing workload and whenever the 95^{th} percentile of response time is over 2 seconds. However, as shown in Figure 8(d), at t=1100sec, the CPU on the Ent running the database saturates. Since the database server is not replicable, increasing capacity of the other two tiers beyond this point does not yield any further increase in effective capacity. However, the black box approach is unaware of where bottleneck lies and continues to add servers to the first two tiers until it has used up all available servers. The response time continues to degrade despite this capacity addition as the Java servlets spend increasingly larger amounts of time waiting for queries to be returned by the overloaded database (see Figures 8(c) and (d)).

We repeat this experiment using our multi-tier provisioning technique. Our results are shown in Figure 9. As shown in Figure 9(b), the control plane adds servers to the application at t=390sec in response to the increased workload. However, beyond this point, no additional capacity is allocated. Our technique correctly identifies that the capacity of the database tier for this workload is around 600 simultaneous sessions. Consequently, when this capacity is reached and the database saturates, it triggers policing instead of provisioning. The admission control is triggered at t=1070sec and drops any sessions in excess of this limit during the remainder of the experiment. Figure 9(d) shows

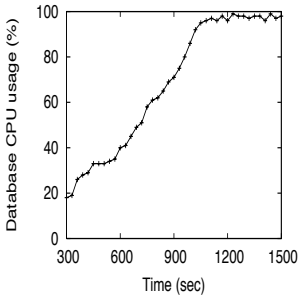


(a) Num servers

(b) Num active sessions



(c) Response time



(d) CPU util at database

Figure 8: Rubis: Blackbox provisioning

that our provisioning is able to maintain a satisfactory response time throughout the experiment.

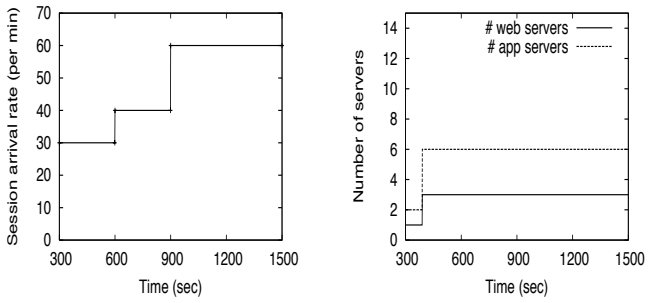
Result: Our provisioning technique is able to take constraints imposed by non-replicable tiers into account. It can maintain response time targets by invoking the admission control when capacity addition does not help.

8.3 Predictive and Reactive Provisioning

In this section we present experiments to demonstrate the need to have *both* predictive and reactive provisioning mechanisms. We used Rubis in these experiments. The workload was generated based on the Web traces from the 1998 Soccer World Cup site [1]. These traces contained the number of arrivals per minute to this Web site over an 8-day period. Based on these we created several smaller traces to drive our experiments. These traces were obtained by compressing the original 24-hr long traces to 1 hr—this was done by picking arrivals for every 24^{th} minute and discarding the rest. This enables us to capture the time-of-day effect as a "time-of-hour" effect. The experiment invoked predictive provisioning once every 15min over the one hour duration and we *refer to these periods as Intervals 1-4*; reactive provisioning was invoked on-demand or once every few minutes. For the sake of convenience, in the rest of the section, we will simply refer to these traces by the day from which they were constructed (even though they are only 1 hr long). Due to lack of space we present only three of these traces: (i) Figure 10(a) shows the workload for day 6 (a typical day), (ii) Figure 11(a) shows the workload for day 7, (moderate overload), and (iii) Figure 12(a) shows the workload for day 8 (extreme overload). Throughout this experiment, we will assume that the database tier has sufficient capacity to handle the peak observed on day 8 and does not become a bottleneck. The average session duration in our trace was 5min.

8.3.1 Only predictive provisioning

Figure 10 presents the performance of the system during day 6 with the control plane employing only predictive provisioning (with reactive provisioning disabled). Day 6 was a "typical" day, meaning the workload closely resembled that observed during the previous days. The prediction algorithm was successful in exploiting this and was able to assign sufficient capacity to the application at all times. In Figure 10(b) we observe that the predicted arrivals closely



(a) Session arrival rate (b) Num servers

(c) Num active sessions (d) Response time

Figure 9: Rubis: Model-based multi-tier provisioning

matched the actual arrivals. The control plane added servers at t=30min—well in time for the increased workload during the second half of the experiment. The application experienced satisfactory response time throughout the experiment (Figure 10(c)).

Result: Our predictive provisioning works well on "typical" days.

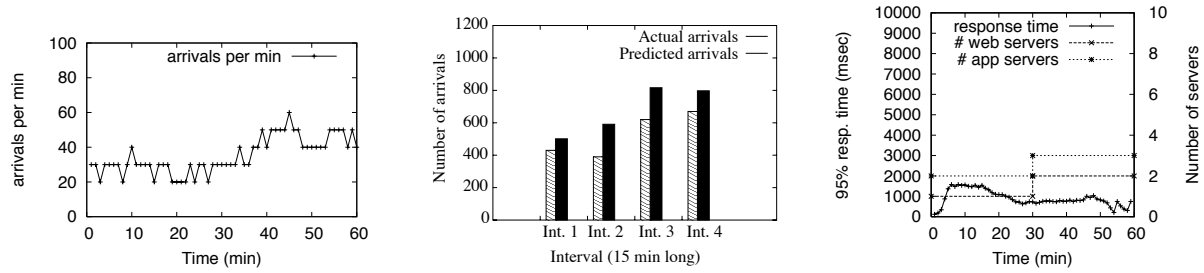
8.3.2 Only reactive provisioning

In Figure 11 we present the results for day 7. Comparing this workload with that on day 6, we find that the application experienced a moderate overload on day 7, with the arrival rate going upto about 150 sessions/min, more than twice the peak on day 6. The workload showed a monotonically increasing trend for the first 40min.

We first let the control plane employ only predictive provisioning. Figure 11(b) shows the performance of our prediction algorithm, both with and without using recent trends to correct the prediction. We find that it severely underestimated the number of arrivals in Interval 2. The use of recent trends allowed it to progressively improve its estimate in Intervals 3 and 4 (predicted arrivals were nearly 80% of the actual arrivals in Interval 3 and almost equal in Interval 4). In Figure 11(c) we observe that the response time target was violated in Interval 2 due to under allocation of servers.

Next, we repeated the experiment with the control plane using only reactive provisioning. Figure 11(d) presents the application performance. Consider Interval 2 first—we observe that, unlike predictive provisioning, the reactive mechanism was able to pull additional servers at t=15min in response to the increased arrival rate, thus bringing down the response time within target. However, as the experiment progressed, the server allocation *lagged behind* the continuously increasing workload. Since reactive provisioning only responded to very recent workload trends, it could not anticipate future requirement well and took multiple allocation steps to add sufficient capacity. Meanwhile, the application experienced repeated violations of SLA during Intervals 2 and 3.

Result: We need reactive mechanisms to deal with large flash crowds. However, reactive provisioning alone may not be effective, since its actions lag the workload.



(a) Session arrivals (b) Actual and predicted arrivals (c) Resp time and num servers

Figure 10: Provisioning on day 6—typical day.

8.3.3 Integrated provisioning

We used the workload on day 8 where the application experienced an *extremely large overload* (Figure 12(a)). The peak workload on this day was an order of magnitude (about 20 times) higher than on a typical day. Figure 12(b) shows how the prediction algorithm performed during this overload. The algorithm failed to predict the sharp increase in the workload during Interval 1. In Interval 2 it could correct its estimate based on the observed workload during Interval 1. The workload increased drastically (reaching upto 1200 sess/sec) during Intervals 3 and 4, and the algorithm failed to predict it.

In Figure 12(c) we show the performance of Rubis when the control plane employs both predictive and reactive mechanisms and session policing is disabled. In Interval 1, the reactive mechanism successfully added additional capacity (at t=8min) to lower the response time. It was invoked again at t=34min (Observe that predictive provisioning was operating in concert with reactive provisioning; it resulted in the server allocations at t=15, 30, 45min). However, by this time (and for the remainder of the experiment) the workload was simply too high to be serviced by the servers available. We imposed a resource limit of 13 servers for illustrative purposes. Beyond this, excess sessions must be turned away to continue meeting the SLA for admitted sessions. The lack of session policing caused response times to degrade during Intervals 3 and 4.

Next, we repeated this experiment with the session policing enabled. The performance of Rubis is shown in Figure 12(d). The behavior of our provisioning mechanisms is exactly like above. However, by turning away excess sessions, the sentry was able to maintain the SLA throughout.

Result: Predictive and reactive mechanisms and policing are all integral components of an effective provisioning technique. Our data center integrates all of these, enabling it to handle diverse workloads.

8.4 VM-based Switching of Server Resources

We present measurements on our testbed to demonstrate the benefits that our VM-based switching can provide. We switch a server from a Tomcat capsule of Rubis to a Tomcat capsule of Rubbos. We compare five different ways of switching a server to illustrate the salient features of our scheme:

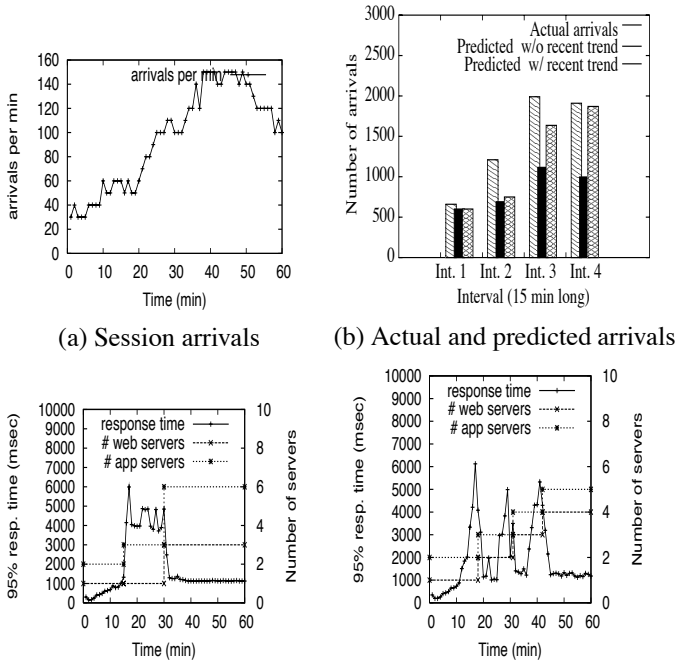
Scenario 1: New server taken from the free pool of servers; capsule and nucleus have to be started on the server.

Scenario 2: New server taken from the free pool of servers; capsule already running on a VM.

Scenario 3: New server taken from another application with residual sessions; we wait for all residual sessions to finish.

Scenario 4: New server taken from another application with residual sessions; we let the two VMs share the CPU equally while the residual sessions exist.

Scenario 5: New server taken from another application with residual sessions; we change the CPU shares of the involved VMs using the "fixed rate ramp down" strategy in Section 6.



(a) Session arrivals (b) Actual and predicted arrivals

(c) Only predictive provisioning (d) Only reactive provisioning

Figure 11: Provisioning on day 7—moderate overload

Scenario	Switching time	r.t. during switching
1	10 ± 1 sec	n/a
2	0	n/a
3	17 ± 2 min	n/a
4	< 1 sec	2400 ± 200
5	< 1 sec	950 ± 100

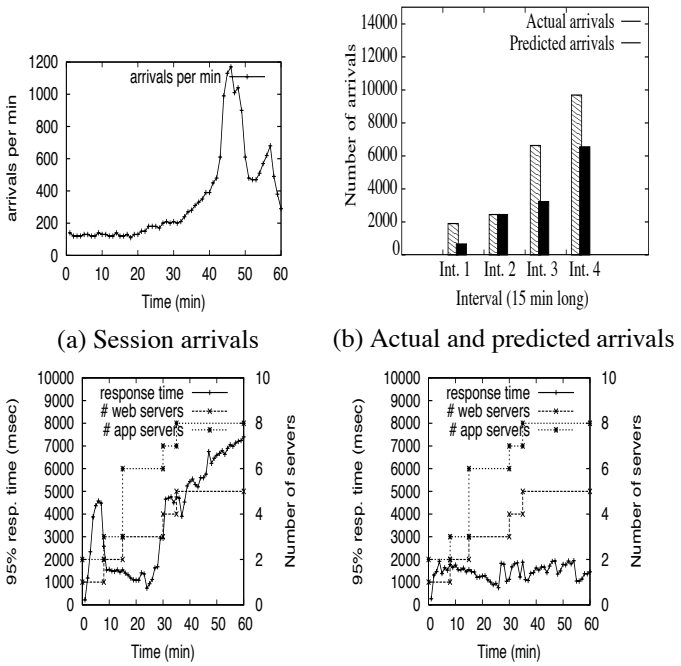
Table 1: Performance of VM-based switching; "n/a" stands for "not applicable".

Table 1 presents the switching time and the performance of residual sessions of Rubis in each of the above scenarios. Comparing scenarios 2 and 3, we find that in our VM-based scheme, the time to switch a server is solely dependent on the residual sessions—the residual sessions of Rubis took about 17min to finish resulting in the large switching time in scenario 3. Scenarios 4 and 5 show that by letting the two VMs coexist while the residual sessions finish, we can eliminate this switching time. However, it is essential to continue providing sufficient capacity to the residual sessions during the switching period to ensure good performance—in scenario 4, new Rubbos sessions deprived the residual sessions of Rubis of the capacity they needed, thus degrading their response time.

Result: Use of virtual machines can enable agile switching of servers. Our adaptive techniques improve upon the delays in switching caused by residual sessions.

8.5 System Overheads

Two sources of overhead in the proposed system are the virtual machines that run on the Elf nodes and the nuclei that run on all nodes. Measurements on our prototype indicate that the CPU overhead and network traffic caused by the nuclei is negligible. The control plane runs on a dedicated node and its scalability is not a cause of concern. We chose the Xen VMM to implement our switching scheme since the performance of Xen/Linux has been shown to be consistently close to native Linux [2]. Further, Xen has been shown to provide good performance isolation when



(a) Session arrivals (b) Actual and predicted arrivals

(c) Provisioning, no policing (d) Provisioning and policing

Figure 12: Provisioning on day 8—extreme overload

running multiple VMs simultaneously, and is capable of scaling to 128 concurrent VMs.

9 Conclusions

In this paper,we argued that dynamic provisioning of multi-tier Internet applications raises new challenges not addressed by prior work on provisioning single-tier applications. We proposed a novel dynamic provisioning technique for multi-tier Internet applications that employs (i) a flexible queuing model to determine how much resources to allocate to each tier of the application, and (ii) a combination of predictive and reactive methods that determine when to provision these resources, both at at large and small time scales. We proposed a novel data center architecture based on virtual machine monitors to reduce provisioning overheads. Our experiments on a forty machine Linux-based hosting platform demonstrate the responsiveness of our technique in handling dynamic workloads. In one scenario where a flash crowd caused the workload of a three-tier application to double, our technique was able to double the application capacity within five minutes while maintaining response time targets. Our technique also reduced the overhead of switching servers across applications from several minutes or more to less than a second, while meeting the performance targets of residual sessions.

References

[1] M. Arlitt and T. Jin. Workload Characterization of the 1998 World Cup Web Site. Technical Report HPL-1999-35R1, HP Labs, 1999.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebuer, I. Pratt, and A. Warfield. Xen and the Art of Virtulization. In *Proceedings of the 19th SOSP*, 2003.

[3] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle. Managing Energy and Server Resources in Hosting Centers. In *Proceedings of the 18th SOSP*, pages 103–116, October 2001.

[4] R. Doyle, J. Chase, O. Asad, W. Jin, and Amin Vahdat. Model-Based Resource Provisioning in a Web Service Utility. In *Proceedings of the 4th USITS*, March 2003.

[5] DynaServer Project. http://compsci.rice.edu/CS/Systems/DynaServer/.

[6] A. Fox, S. Gribble, Y. Chawathe, E. Brewer, and P. Gauthier. Cluster-based Scalable Network Services. In *Proceedings of the 16th SOSP*, December 1997.

[7] 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.

[8] A. Kamra, V. Misra, and E. Nahum. Yaksha: A Controller for Managing the Performance of 3-Tiered Websites. In *Proceedings of the 12th IWQoS*, 2004.

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

[10] Kernel TCP Virtual Server. http://www.linuxvirtualserver.org/software/ktcpvs/ktcpvs.html.

[11] Oracle9i. http://www.oracle.com/technology/products/oracle9i.

[12] V. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Proceedings of the 8th ASPLOS*, October 1998.

[13] J. Rolia, X. Zhu, M. Arlitt, and A. Andrzejak. Statistical Service Assurances for Applications in Utility Grid Environments. Technical Report HPL-2002-155, HP Labs, 2002.

[14] Y. Saito, B. Bershad, and H. Levy. Manageability, availability and performance in porcupine: A highly scalable, cluster-based mail service. In *Proceedings of the 17th SOSP*, 1999.

[15] Sysstat package. http://freshmeat.net/projects/sysstat.

[16] K. Shen, H. Tang, T. Yang, and L. Chu. Integrated Resource Management for Cluster-based Internet Services. In *Proceedings of the Fifth OSDI*, December 2002.

[17] B. Urgaonkar, P. Shenoy, and T. Roscoe. Resource Overbooking and Application Profiling in Shared Hosting Platforms. In *Proceedings of the Fifth OSDI*, December 2002.

[18] 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.

[19] M. Welsh and D. Culler. Adaptive Overload Control for Busy Internet Servers. In *Proceedings of the 4th USITS*, March 2003.

[20] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the 18th SOSP*, October 2001.