SLA Decomposition: Translating Service Level Objectives to System Level Thresholds

Yuan Chen, Subu Iyer, Xue Liu, Dejan Milojicic, Akhil Sahai
HP Labs
{firstname.lastname}@hp.com

Abstract

In today's complex and highly dynamic computing environments, systems/services have to be constantly adjusted to meet Service Level Agreements (SLAs) and to improve resource utilization, thus reducing operating cost. Traditional design of such systems usually involves domain experts who implicitly translate Service Level Objectives (SLOs) specified in SLAs to system-level thresholds in an ad-hoc manner. In this paper, we present an approach that combines performance modeling with performance profiling to create models that translate SLOs to lower-level resource requirements for each system involved in providing the service. Using these models, the process of creating an efficient design of a system/service can be automated, eliminating the involvement of domain experts. We demonstrate that our approach is practical and that it can be applied to different applications and software architectures. Our experiments show that for a typical 3-tier e-commerce application in a virtualized environment the SLAs can be met while improving CPU utilization up to 3 times.

1. Introduction

A Service Level Agreement captures the formal agreement between a service provider and one of its customers relating to service behavior guarantees, escalation procedures, and penalties in case the guarantees are violated. When an SLA requirement is agreed upon, a service administrator usually designs the service and then stages it. In many cases, the staging process is iterative in nature and involves several manual interventions. Once the service is observed to behave satisfactorily during staging, it is put in production.

Enterprise applications and services are typically comprised of a large number of components, which interact with one another in a complex manner. Since each sub-system or component potentially affects the overall behavior of the system, any high level goal (e.g., performance, availability, security, etc.) specified for the service potentially relates to all low-level sub-systems or components. One of the key tasks during the design stage is SLA decomposition — deriving low level system thresholds from Service Level Objectives (SLOs) specified in SLAs. The thresholds can then be used to create an efficient design to meet the SLA. For example, the system thresholds are used to determine how much and how many of the resources should be allocated to satisfy the proposed SLA requirement. With the advent of virtualization and application sharing techniques, opportunities exist for improving overall system performance and resource utilization by allocating optimal resources for the service.

System administrators and experts normally apply their domain knowledge to implicitly map high level goals to lower level metrics, i.e., use past experience with specific applications to determine low level thresholds necessary to ensure that the overall system goals are met. Automatically deriving and inferring low level thresholds from high level goals are difficult tasks due to the complexity and dynamism inherent in such systems. The range of design choices in terms of operating systems, middleware, shared infrastructures, software structures etc. further complicates the problem. For example, different virtualization technologies (e.g., Xen [2] or VMware [4]) can be used in a utility data center. Applications can use different software structures (e.g., 2-tier PHP, 3-tier Servlet, or 3-tier EJB [14]) to implement the same functionality. Different implementations are also available for each tier (e.g., Apache or IIS for web server; WebLogic, WebSphere, or JBoss for EJB server; Microsoft SQL Server, Oracle, or MySQL as database server).

In our work, we propose a general methodology that combines performance modeling and profiling to accomplish SLA decomposition. The intent is to first model and characterize the behavior of a service and then to use this model to predict the required design of a service instance with different high level goals and configurations. While the high-level SLA requirements

Fourth International Conference on Autonomic Computing (ICAC'07)
0-7695-2779-5/07 $20.00 © 2007 IEEE

may include performance, availability, security, etc., we focus on performance goals in this paper.

We provide a general approach for calculating the bounds on system behavior given performance-oriented SLOs for the service. Our approach uses analytical models to capture the relationship between high level performance goals (e.g., response time of the overall system) and the refined goals for each component (e.g., average service time of each component). In particular, we present a novel queueing network model for multi-tier architecture, where each tier is modeled as a multi-station queueing center. Our model is sufficiently general to capture a number of commonly used multi-tier applications with different application topology, configuration, and performance characteristics. Our approach also builds profiles characterizing per-component performance metrics (e.g., average service time) as functions of resource allocations (e.g., CPU, memory) and configuration parameters (e.g., max connections). With the analytical models and the component profiles, the low level operational goals can be derived by translating high level performance goals to component level goals and using the profiles to determine component level resource requirements and configuration, which can meet high level goals. The low level goals can then be used to create an efficient design to meet the high level SLA. Some of the thresholds, such as healthy ranges of lower level metrics, are used for monitoring the systems during operation. The developed models are archived for future reuse, both analytical models and component profiles.

The remainder of this paper is organized as follows. Section 2 describes a motivating scenario for SLA decomposition in a virtualized data center. Section 3 provides an overview of our approach. We then describe in detail a novel analytical performance model for multi-tier applications in Section 4. Section 5 presents the implementation of profiling and decomposition of a multi-tier application as the experimental validation of our approach. Related work is discussed in Section 6. Section 7 concludes the paper and discusses future work.

2. Motivating Scenario

Today's enterprise data centers are designed with on-demand computing and resource sharing in mind, where all resources are pooled into a common shared infrastructure [3]. Virtualization technologies such as VMware ESX Server [4] and Xen Virtual Machine Monitor [2] enable applications to share computing resources with performance isolation. Such a model also allows organizations to flex their computing resources based on business needs. Typically, such data centers host multiple applications (often from different customers).

Consider a typical 3-tier application consisting of a web server, an application server and a database server in the virtualized data center, where each tier is hosted on a virtual machine. Figure 1 shows the application's average response time with three different CPU shares assigned to the virtual machine hosting the application server tier (i.e. Tomcat). Given the SLO of average response time less than 10 seconds, the configuration with CPU assignment of 20% fails to meet the SLO while the CPU assignment of 90% meets the SLO but the system is over-provisioned since CPU assignment of 50% is sufficient to ensure the SLO. One key task of designing such a system is to determine the resource requirement of each tier to meet high level SLA goals while achieving high resource utilization. For the above example, SLA decomposition determines the CPU assignment to Tomcat, e.g., "CPU assignment = 50%" such that if the virtual machine is configured that way, the application will meet the response time requirement with reasonable CPU utilization.

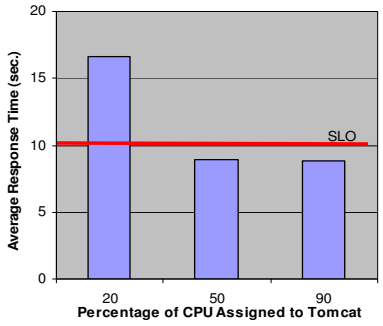

Figure 1. Performance of a multi-tier application in a virtualized data center

3. SLA Decomposition

Given high level goals, SLA decomposition translates these goals into bounds on low level system metrics such that the high level goals are met. In other words, the task of SLA decomposition is to find the mapping of overall service level goals (e.g., SLOs) to the state of each individual component involved in providing the service (e.g., resource requirement and configuration). For example, given SLOs of a typical 3-tier e-commerce environment in terms of response time and throughput requirement, the decomposition task is to find the following mapping

$(R, T) \rightarrow (\theta_{http\text{-}cpu}, \theta_{http\text{-}mem}, \theta_{app\text{-}cpu}, \theta_{app\text{-}mem}, \theta_{db\text{-}cpu}, \theta_{db\text{-}mem})$

where R and T denote the response time and throughput of the service respectively and θ is the resource requirement. SLA decomposition problem is the opposite of a typical performance modeling problem, where the overall system's performance is predicted based on the configuration and resource consumption of the sub-components.

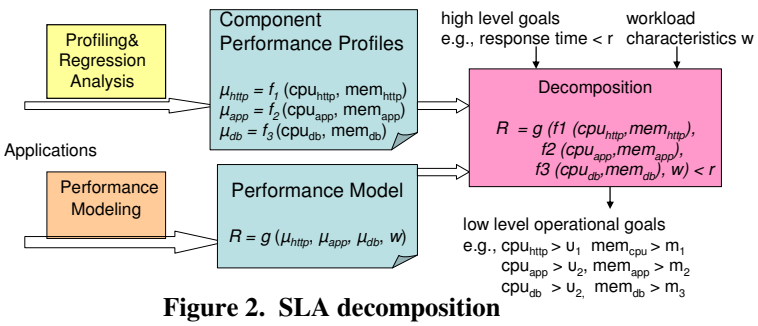

Figure 2. SLA decomposition

The conceptual architecture of our approach is illustrated in Figure 2. We benchmark the application and generate a detailed performance profile for each component. Analytical model is built to capture the relationship between the application's high level goals (e.g., application performance) and lower level goals (e.g., single component's performance and configuration). We then use the profile and analytical models to generate low level operational goals by decomposition.

3.1 Component Profiling

This step creates detailed profiles of each component. A component profile captures the component's performance characteristics as a function of the resources that are allocated to the component and its configuration. In order to obtain a component profile, we deploy a test environment and change the resources (e.g., CPU, memory) allocated to each component. We then apply a variety of workloads and collect the component's performance characteristics independent of other components (e.g, mean service rate μ and variance of service time σ). After acquiring the measurements, general functional mappings from system metrics to the component's performance metrics are derived using either a classification or regression analysis based approach. For example, Apache Web server's profile captures the correlation between an Apache Web server's mean service rate and the CPU and memory allocated to it, i.e. $\mu = f(CPU, MEM)$. The profiling can be performed either through operating system instrumentation [13] or estimation based on application or middleware's monitoring information [17] (e.g., service time recorded in Apache and Tomcat log file). The former approach can achieve transparency to the application and component middleware but may involve changes to the system kernel while the latter approach is less intrusive.

3.2 Performance Modeling

Performance modeling captures the relationship between each single component and the overall system performance. For example, given performance characteristics of each of the components in a 3-tier application, μ_{http}, μ_{app}, μ_{db}, and the workload characteristics of the overall system w, model $R = g(\mu_{http}, \mu_{app}, \mu_{db}, w)$ predicts the response time of the 3-tier application. We propose a novel queueing network model of multi-tier applications. In this model, the server at each tier is modeled as a multi-station queueing center (i.e., *G/G/K* queue) which represents the multi-threaded architecture commonly structured in the modern servers (e.g., Apache, Tomcat, JBoss, and MySQL). An application with N tiers is then modeled as a closed queueing network of *N* queues $Q_1, Q_2, \ldots Q_N$. Each queue represents a tier of the application and the underlying server that it runs on. Mean-value analysis (MVA) [5] is used for evaluating the performance of the queueing network. Such a model can handle user-sessions based workloads found in most e-business applications and accurately predict the multi-tier application's performance based on single tier's performance [5] and the workload characteristics, such as the flow of the requests across tiers. Our model can handle arbitrary service rate distribution as well as multiple visits to a tier. Since we explicitly capture the concurrent limits in our model (e.g., max number of concurrent threads), this model inherently handles concurrent limits at tiers. The performance model is further discussed in Section 4.

3.3 Decomposition

Once we have the component profile, $\mu_{http} = f_1(CPU_{http})$, $\mu_{app} = f_2(CPU_{app})$, $\mu_{db} = f_3(CPU_{db})$, and the model $R = g_1(\mu_{http}, \mu_{app}, \mu_{db}, w)$ and $T = g_2(\mu_{http}, \mu_{app}, \mu_{db}, w)$, the decomposition of high level goals response time $R < r$ and throughput $X > x$ is to find the set of CPU_{http}, CPU_{app}, CPU_{db} satisfying the following constraints:

$$g_1(f_1(CPU_{http}), f_2(CPU_{app}), f_3(CPU_{db}), w) < r$$

$$g_1(f_1(CPU_{http}), f_2(CPU_{app}), f_3(CPU_{db}), w) > x$$

Other constraints, such as "minimize CPU_{http}+ CPU_{app}+ CPU_{db}", can also be added.

Once the equations are identified, the decomposition problem becomes a constraint satisfaction problem. Various constraint satisfaction algorithms, linear programming and optimization techniques are available to solve such problems [20]. Typically, the solution is non-deterministic and the solution space is large. However, for the problems we are studying, the search space is relatively small. For example, if we consider assigning CPU to virtual machines at a granularity of 5%. We can efficiently enumerate the entire solution space to find the solutions. Also, we are often interested in finding a feasible solution, so we can stop the search once we find one. Other heuristic techniques can also be used during the search. For example, the hint that the service time of the component typically decreases with respect to the increase of resource allocated to it can reduce the search space.

If the high level goals or the application structures change, we only need to change the input parameters of analytical models and generate new low level operational goals. Similarly, if the application is deployed to a new environment, we only need to regenerate a profile for new components in that environment. Further, given high level goals and resource availability, we can apply our decomposition approach for automatic selection of resources and for generation of sizing specifications that could be used during system deployment. The generated thresholds can be used for creating efficient designs and for monitoring systems for proactive assessment of SLOs. The detailed implementations of modeling, profiling and decomposition of multi-tier applications in a virtual data center are discussed in the following two sections.

4. Modeling Multi-Tier Web Applications

4.1 Basic Queueing Network Model

Modern Web applications and e-Business sites are usually structured into multiple logical tiers, responsible for distinct set of activities. Each tier provides certain functionality to its preceding tier and uses the functionality provided by its successor to carry out its part of the overall request processing. Consider a multi-tier application consisting of M tiers, $T_1, \ldots, T_M$. In the simplest case, each request is processed exactly once by each tier and forwarded to its succeeding tier for further processing. Once the result is processed by the final tier T_M, the results are sent back by each tier in the reverse order until it reaches T_1, which then sends the results to the client. In more complex processing scenarios, each request at tier T_i, can trigger zero or multiple requests to tier T_{i+1}. For example, a static web page request is processed by the Web tier entirely and will not be forwarded to the following tiers. On the other hand, a keyword search at a Web site may trigger multiple queries to the database tier.

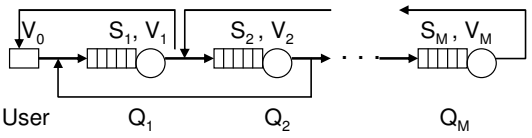

Figure 3. Basic queueing network model

Given an M-tier application, we model the application using a network of M queues $Q_1, Q_2, \ldots, Q_M$ (see Figure 3). Each queue represents an individual tier of the application. Each queue models the request queue on the underlying server where it runs on. A request, after being processed at queue Q_i either proceeds to Q_{i+1} or returns to Q_{i-1}. A transition to the client denotes a request complementation (i.e. response to the client). We use V_i to denote the average request rate serviced by Q_i. Our model can handle multiple visits to a tier. Given the mean service time S_i of queue Q_i, the average service demand per user request D_i at Q_i can be approximated as $S_i \times V_i / V_0$, where V_0 is average request rate issued by the users.

4.2 Multi-Station Queueing Network Model

Modern servers typically utilize a multi-thread and/or multi-process architecture. The server listens in the main thread for requests. For each request, it allocates a thread to handle it. For example, the flow of servicing a static HTTP request is as follows. A request enters the TCP accept queue where it waits for a worker thread. A worker thread processes a single request to completion before accepting another new request. In the most general case, each of the tiers may involve multiple servers and/or multiple threads. The application server tier for example may involve one or more multi-threaded application servers (e.g., Tomcat) running on multiple processors. A similar notion is applicable to the database tier which may consist of one or more database servers (e.g., MySQL) which in turn may run on a multi-threaded/multi-processor system.

The amount of concurrency may also be determined by the number of processes or concurrent threads/servers the tier supports. In order to capture the multi-thread/server architecture and the concurrency, we enhance the basic model by using a multi-station queueing center to model each tier. In this model, each worker thread/server in the tier is represented by a station. The multi-station queueing model thus is the general representation of a modern server architecture.

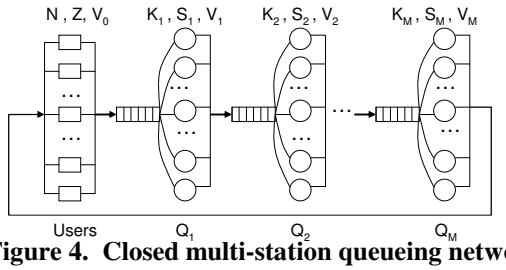

Figure 4. Closed multi-station queueing network

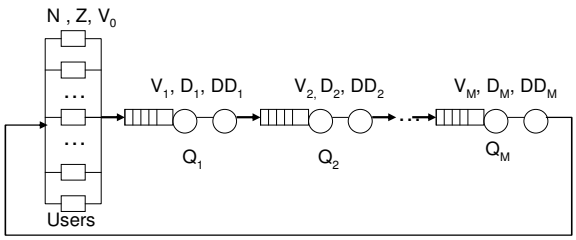

Figure 5. Approximate model for MVA analysis

4.3 Closed Multi-Tier Multi-Station Queueing Networ Network Model

The workload on a multi-tier application is typically user session-based, where a user session consists of a succession of requests issued by a user with think time *Z* in between. At a time, multiple concurrent user sessions interact with the application. In order to capture the user session workload and the concurrency of multiple sessions, we use a closed queueing network, where we model concurrent sessions by *N* users in the queueing system. Figure 4 shows the closed multi-station queueing network model (QNM) of a multi-tier application. Each tier is modeled by a multi-station queueing center as discussed earlier, with the number of stations being the server's total number of worker threads.

We use K_i to denote the number of worker threads at tier *i*. Similarly, the mean service time at tier *i* is denoted by S_i. A user typically waits until the previous request's response returns to send the following request. The average time elapsed between the response from a previous request and the submission of a new request by the same user is called the "think time", denoted by *Z*.

4.4 Deriving Queueing Network Performance

Given the parameters *{N, Z, V_i, M_i, S_i}*, the proposed closed queueing network model can be solved analytically to predict the performance of the underlying system. For example, an efficient algorithm such as the Mean-value analysis (MVA) can be used to evaluate the closed queueing network models with exact solutions [5]. MVA algorithm is iterative. It begins from the initial conditions when the system population is *1* and derives the performance when the population is *i* from the performance with system population of *(i-1)*, as follows

$$R_k(i) = \begin{cases} D_k & \text{delay resource} \\ D_k \times (1 + Q_k(i-1)) & \text{queueing resource} \end{cases}$$

$$X(i) = \frac{i}{\sum_{k=1}^{K} R_k(i)}$$

$$Q_k(i) = X \times R_k(i)$$

where $R_k(i)$ is the mean response time (mean residence time) at server *k* when system population is *i*; $R_k(i)$ includes both the queueing time and service time; *X(i)* the total system throughput when system population is *i*; and $Q_k(i)$ is the average number of customers at server *k* when system population is *i*.

Traditional MVA has a limitation that it can only be applied to single-station queues. In our model, each tier is modeled with a multi-station queueing center. To solve this problem, we adopt an approximation method proposed by Seidmann et al. [6] to get the approximate solution of performance variables. In this approximation, a queueing center that has *m* stations and service demand D[1] at each station is replaced with two tandem queues. The first queue being a single-station queue with service demand *D/m*, and the second queue is a pure delay center, with delay *D×(m-1)/m*. It has been shown that the error introduced by this approximation is small [7]. By using this approximation, the final queueing network model is shown in Figure 5 where qrD_i and drD_i are average demands of the regular queueing resource and the delay resource in the tandem queue respectively.

The modified MVA algorithm used to solve our queueing network is presented in Figure 6. The algorithm takes the following set of parameters of a multi-tier application as inputs:

N: number of users;
Z: think time;
M: number of tiers;
K_i: number of stations at tier *i (i = 1,..., M);*
D_i: service demand at tier *i (i = 1,..., M);*
V_i : mean request rate of tier *i (i = 1,..., M);*

The MVA algorithm computes the average response time R and throughput X of the application.

4.5 Model Validation

To validate the correctness and accuracy of our model, we experimented with two open-source 3-tier applications running on a virtualized Linux-based server testbed. The testbed is composed of four machines. One

[1] D_i represents the average service demand per user request at Q_i . It can be approximated as $S_i \times V_i / V_0$.

Input: $N, Z, M, K_i, S_i, Vi\ (i = 1,.. M)$
Output: R, X
//initialization
$R_0 = Z;\ D_0 = Z;\ Q_0 = 0;$
for $i = 1$ to M {
// Tandem approximations for each tier
$Q_i = 0;\ D_i = (S_i\ 0\ V_i) / V_0;$
$qrD_i = D_i/K_i;\ \ drD_i = D_i \times (K_i\text{-}1)/K_i;$ }
//introduce N users one by one
for i = 1 to N {
for $j = 1$ to M {
$R_j = qrD_j \times (1 + Q_j);$ // queueing resource
$RR_j = drD_j;$ } //delay resource
for $j = 1$ to M
$Q_j = X \times R_j;$
$X = \dfrac{i}{R_0 + \sum_{j=1}^{m}(R_i + RR_i)}$
}
$R = \sum_{i=1}^{M}(R_i + RR_i)$

Figure 6. Modified MVA algorithm

of them is used as client workload generator and the other three machines are used as Apache 2.07 web server, Tomcat 5.5 Servlet server and MySQL 5.0 database server respectively. We measure the service time for each tier by computing the elapsed time when a thread is dispatched to process a new request at that tier and when it finishes the task. Another required parameter for our model is the number of stations for each queue or tier. For Apache and Tomcat, the total number of stations is determined by the size of thread pool (i.e., maxClients in Apache and maxThreads in Tomcat). MySQL manages threads in a more dynamic fashion. Depending on the server configuration settings and current status, the thread may be either created new, or dispatched from the thread cache. The average number of all worker threads during a run is used to approximate the number of stations. This approximate model enables us to use load-independent multi-station queueing to model thread cache based server. Average visit rate of each tier is obtained from log files.

The first application we use is TPC-W [19], an industry standard e-commerce application. TPC-W specifies 14 unique Web interactions. The database is configured for 10,000 items and 288,000 customers. Session based workload is generated from a client program to emulate concurrent users. The think time is set to 0.035 seconds. The max clients of Apache and max threads of Tomcat are set as 50. We change workload by varying the number of concurrent sessions

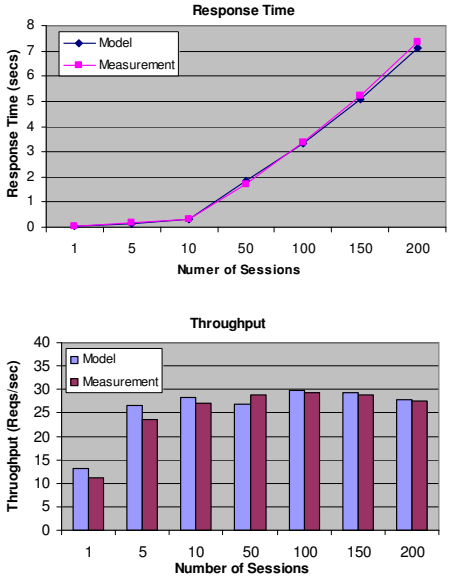

Figure 7. TPC-W performance

generated by the workload generator. Each run lasts 200 seconds after 60 seconds of warm-up period. We measure different model input and output parameters during each run. We then apply MVA algorithm described in section 4.4 to derive the response time and throughput. Figure 7 shows the results of the response time and throughput predicted by the model and the response time directly measured for number of concurrent sessions varying from 1 to 200. From the figures, we can see that the analytic model does predict the performance of TPC-W accurately. The results predicted by our model are close to the measurement under different workloads, even when the application reaches its maximum throughput.

To further validate the effectiveness of our performance model, we experimented with RUBiS, an eBay like auction site developed at Rice University [1]. It defines 26 interactions and has 1,000,000 users and 60,000 items. The think time is exponential distribution with a mean of 3.5 seconds. We vary the number of concurrent sessions from 50 to 300 and each run lasts 300 seconds with a 120 seconds warm up. Unlike the TPC-W experiments, we use the same set of input parameters obtained during profiling to predict the performance for different workloads. The results of response time and throughput are depicted in Figure 8. Even using the same set of model input parameters, the model can still accurately predict the performance for different workloads.

5. Profiling and SLA Decomposition

5.1 Experimental Infrastructure

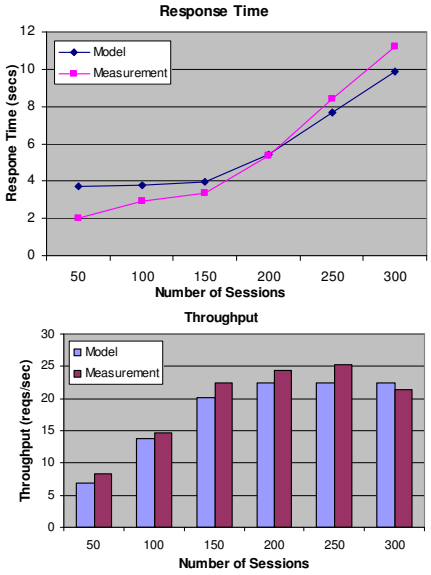

Figure 8. RUBiS performance

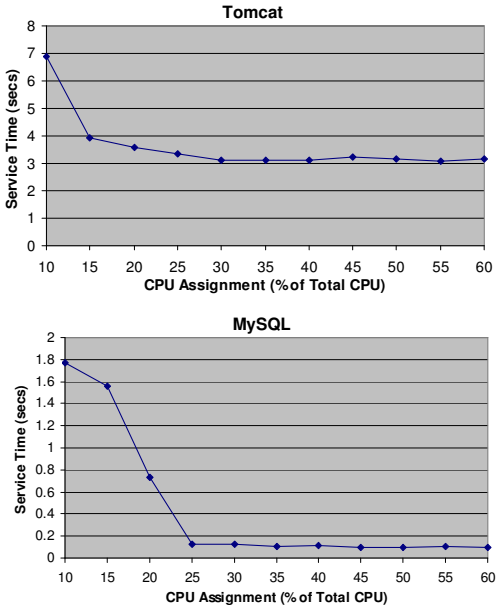

Figure 9. Service Time Profiles

Our experimental testbed consists of a virtualized data center where multiple applications share a common pool of resources. We use a cluster of dual processor x86 based servers with Xen virtual machines (VMs) to simulate such a virtual data center. The testbed consists of multiple HP Proliant servers, each running Fedora 4, kernel 2.6.12, and Xen 3.0-teseting. Each of the server nodes has two processors, 4 GB of RAM, and 1G Ethernet interfaces. These hardware resources are shared between the virtual machines that host the application. Each virtual machine can be instantiated from a bunch of VM images, database images, and swap images. These hardware resources are shared between the virtual machines that host the application.

5.2 Building Profile

For the profiling, we use a 3-tier Servlet based implementation of RUBiS [1] consisting of an Apache Web server 2.0, a Tomcat 5.5 servlet container, and a MySQL 5.0 database server, running on virtual machines hosted on different servers. A synthetic workload generator runs on the fourth server. To isolate performance interference, we restrict the management domain to use one CPU and virtual machines to use the other CPU.

One of the key objectives of profiling is to accurately estimate the service time of each component since the accuracy of a model depends directly on the quality of its input parameters. To effectively measure service time for each component, the time stamp is recorded either when a new thread is created or when an idle thread is assigned. Similarly, we record the timestamp when the thread is returned to the thread pool or destroyed. The time interval between the two time stamps is the time spent in each component. This time also includes the waiting time for its neighbor's reply. The time spent on waiting for next tier is measured in a similar way. The difference between the two time intervals is the actual service time. This approach works for both lightly loaded as well as overloaded systems. Details of measurement implementation can be found in [19].

During profiling, we also collect the workload characteristics, including the average visit rate on each tier V_i. This number is used to derive the average service demand D_i per user request such as $D_i = V_i/V_0 * S_i$, where S_i is the mean service time and V_0 is the average user request rate.

For the purpose of profiling, we systematically change the configurations of each virtual machine hosting the application including assignment of resources (e.g., CPU, MEM) to each virtual machine and software configuration parameters (e.g., maximum number of clients, thread cache size, query cache size, etc.). We then apply certain workloads to each tier and measure the service time of that tier. When we profile a tier, we configure other tiers at its maximum capacity to prevent them from becoming performance bottlenecks. This ensures that interdependencies do not affect the accuracy of the profile. After collecting the service times for different configurations, we apply statistical analysis techniques, such as regression analysis, to

Table 1. 3-tier RUBiS

SLOs	Design	CPU Assignment			Performance		CPU Utilization	
		Tomcat	MySQL	Total	Resp (sec)	Thrpt (rqs/s)	Tomcat	MySQL
Users=300 Res<10sec Thrpt>20 reqs/sec	Optimal	40%	25%	65%	9.79	21.41	67%	70%
	System0	**45%**	**30%**	**75%**	**9.89**	**21.59**	**61%**	**67%**
	System1	20%	20%	40%	15.93	11.1	99%	92%
	System2	90%	90%	180%	8.86	24.3	23%	21%
Users=100 Resp.<5sec Thrpt>10rqs/s	**System0**	**15%**	**15%**	**30%**	**4.83**	**13**	**74%**	**69%**

derive correlation between the service times of a tier and its respective configuration.

Using the steps mentioned above, we have built profiles for Apache, Tomcat and MySQL with different CPU assignments to each tier. We used SEDF (Simple Earliest Deadline First) algorithm [2] that Xen provides for controlling the percentage of total CPU assigned to a virtual machine. We used the capped mode of SEDF to enforce the fact that a virtual machine cannot use more than its share of the total CPU. We changed the CPU assignment from 10% to 100% to measure the service times with different CPU assignments.

Figure 9 shows the service time of Tomcat and MySQL as a function of the percentage of CPU assignment to the virtual machine hosting them. As shown in the figure, both Tomcat and MySQL demonstrate similar behavior. As the CPU assignment increases, the service time drops initially and remains constant after getting enough CPU. The results are then saved as Tomcat and MySQL's profiles.

5.3 SLA Decomposition and Validation

Our performance model can be represented as follows:

$$R = g(N, Z, M, K_i, S_i, V_i)$$

$$X = N/(R+Z)$$

where variables R, X and N denote response time, throughput and the number of concurrent users respectively. Please see Section 4 for the definitions of the other variables. We also obtain the service time profile $S_i = f_i(CPU_i)$, the number of stations K_i for each component and the workload characteristics, such as average visiting rate V_i and think time Z via profiling.

Given high level goals of R < r, X > x and N of a M-tier application, the decomposition problem is to find a set of CPU_i (i = 1, …M) that satisfies the following constraints:

$g(N, Z, K_1, ...K_m, f_i(CPU_i), f_M(CPU_M), V_1, ... V_M) < r.$

$N / (R+Z) > x$

To find the solution to the above equation, we simply enumerate all combinations of CPU assignments, CPU_i= 10% to 100%, in 10% unit increments that satisfy the constraints. We then choose the combinations of CPU_i such as the sum of CPU_i (i = 1, …, M) is minimized. An advanced algorithm [20] can be applied for complex and large scale systems.

Table 2. 2-tier RUBiS

SLOs	Design	CPU Assignment			Performance		CPU Utilization	
		Apache	MySQL	Total	Resp (sec)	Thrpt (rqs/s)	Apache	MySQL
Users=100 Resp.<5s Thrpt>10rqs/s	System0	10%	15%	25%	4.83	13	72%	53%
Users=500 Resp.<10s Thrpt>40rqs/s	System0	35%	30%	65%	8.2	42	76%	61%

We apply SLA decomposition to design RUBiS systems with different SLO goals and software architectures. Given high level SLO goals, we generate low level CPU requirements through SLA decomposition and then configure the VMs based on the derived low level CPU requirements. We then validate our design by measuring the actual performance of the system and compare the results with the SLA goals. In the experiments, we consider the high level SLA goals defined as number of concurrent users, average response time, and maximum throughput. We use 5% of the total CPU capacity as a unit for CPU assignments.

In the first experiment, we use a 3-tier implementation of RUBiS, an Apache web Server, a Tomcat server and a MySQL database server hosted on VMs on different servers. Table 1 summarizes the results of different CPU assignments for two different SLA goals. The first column shows the SLO goals. The column of *CPU assignment* describes the system design parameters in terms of the percentage of CPU assigned to each tier. The columns of *Performance* and *CPU utilization* show the measured response time, throughput and the CPU utilization of the actual system.

For the SLA goal of *300 users, response time < 5 seconds and throughput > 20 requests/seconds,* optimal system ensures the SLA using the minimum CPU resource (65% of total CPU assigned to Tomcat and MySQL). System0 is the system designed based on the proposed SLA decomposition approach and the assignment of CPU 75% is close to the optimal solution 60%. System0 meets the SLOs with reasonable CPU utilization, i.e., 61% and 67%. Two other systems (system1 and system2) are used for the purpose of comparisons, too. System1 is underprovisioned while System2 is overprovisioned. From the table, we observe that System1 fails to meet the SLA since the system is completely overloaded while system2 meets the SLOs but is highly under-utilized with less than 25% CPU utilization of both Tomcat and MySQL. Compared with the over-provisioning System1, our system (System0) can meet the SLAs using less CPU resource (75% as

opposed to 180%) and improves the utilization 3 times (60% as opposed to 20%).

We also experimented with a less demanding SLA of *100 users, response time < 5 seconds and throughput > 10 requests/seconds*. These results are also summarized in Table 1. From the results, we can see that System0, which is designed based on the low level system thresholds derived by our approach, can meet the high level SLA with efficient CPU utilization.

In order to further check the applicability of our approach, we applied the decomposition to design a 2-tier RUBiS implementation consisting of an Apache web Server and a MySQL database server. The 2-tier application runs PHP script at Web server tier and puts much higher load on web tier than 3-tier. We evaluated our approach with two different SLAs. The results in Table 2 show that our approach can be effectively applied to design such a 2-tier system with different SLA requirements.

6. Related Work

Previous studies have utilized performance models to guide resource provisioning and capacity planning [16], [20]. Urgaonkar et al. propose a dynamic provisioning technique for multi-tier applications [16]. Our work is different from theirs in several aspects. First, their model only takes into account the request rate and number of servers at each tier while our model can estimate how performance is affected by different workloads, resource allocations, and system configurations and can handle general SLOs, such as response time, throughput and the number of concurrent users. Second, they assume an open queueing network for request-based transactions whereas we assume a closed network for user session based interactions. Third, our approach has been applied in virtualized environments, managing resource assignment in a more fine-grained manner than just determining the number of servers for each tier.

Zhang et al. present a nonlinear integer optimization model for determining the number of machines at each tier in a multi-tier server network [20]. The techniques to determine the bounds can be applied to solve our general decomposition problem.

Stewart et al. present a profile-driven performance model for multi-component online service [18]. Similar to ours, their approach builds profiles per component and uses the model to predict average response time and throughput. However, the basic assumption and the focus are different. They use the model to discover component placement and replication that achieve high performance in a cluster-based computing environment while our work is focused on ensuring SLAs are met with optimized resource usage. They profile component resource consumption as a function of different workloads while we profile the component performance characteristics as a function of low level goals such as resource assignments and configurations. As a result, our approach can support a more fine grained resource share and management. Additionally, their approach uses a simple M/G/1 queue to model service delay at each server, which is less accurate than general G/G/1 closed queueing network we used. Though our approach can incorporate different resources, we have not taken into account the I/O and memory profile as they did. Their model explicitly captures communication overhead which is not included in our current model.

A lot of research efforts have been undertaken to develop queueing models for multi-tier business applications. Many such models concern single-tier Internet applications, e.g., single-tier web servers [9], [10], [11], [12]. A few recent efforts have extended single-tier models to multi-tier applications [16], [17], [19]. The most recent and accurate performance model for multi-tier applications is proposed by Urgaonkar [17]. Similar to our model, their model uses a closed queueing network model and mean value analysis (MVA) algorithm for predicating performance of multi-tier applications.[2] Despite the similarities, the two models are different in the following aspects. First, our model uses multi-station queues to capture the multi-thread architecture, hence explicitly handling the concurrency limits. Use of multi-station queues also enables us to model multi-server tier the same way as single server tier. The approximate MVA algorithm for multi-station queue is more accurate than simply adjusting the total workload. Second, our measurement methodology can work well for both light load as well as heavily load conditions. Finally, we systematically study the performance and validate our models in a virtualized environment. These unique features enable us to model the application in a more fine-grained manner and handle various workload conditions in a consistent way. Though our model can be adjusted to handle imbalance across tier replicas and multiple session classes based on queueing theory, we have not explored these areas yet. An early result of our performance model was presented at [19]. The model used in this work makes numerous enhancements to our earlier model, including general model for any tier applications, an integrated MVA analysis with SLA decomposition and additional new experiments for model validation.

[2] The two models were developed concurrently and an early report of our model appeared at [19].

Kelly et al. present an approach to predicting response times as a function of workload [13]. The model does not require knowledge of internal application component structure and uses only transaction type information instead. It has been shown that the approach works well for realistic workload under normal system load, but it's not clear how well it will perform under high system load, which is crucial for our work.

The specific performance model used in this paper is based on queueing model. Conceptually, any model which can help determine the performance (e.g., response time) of applications can be incorporated into our solution. It would be interesting to investigate how to integrate other models into our solution.

7. Conclusion and Future Work

It is a prerequisite for next generation data centers that computing resources are available on-demand and that they are utilized in an optimum fashion. One of the most important steps towards building such systems is to automate the process of designing and thereafter monitoring systems for meeting higher level business goals. These are intriguing but difficult tasks in IT automation. In this paper, we propose SLA decomposition approach that combines performance modeling with performance profiling to solve this problem by translating high level goals to more manageable low-level sub-goals. These sub-goals feature several low level system and application level attributes and metrics that are used for creating an efficient design to meet high level SLAs. We have built a testbed to validate our methodology using a number of multi-tier business applications. The evaluation results show the efficacy of our approach.

In the future, we will investigate how the current approach can be generalized to support different enterprise applications and composed web services running on heterogeneous platforms. We are also planning to look at other system and application level metrics in addition to CPU usage.

8. References

[1] Rubis Rice University Bidding System, http://www.cs.rice.edu/CS/Systems/DynaServer/rubis.

[2] P. Barham, et al. "Xen and the Art of Virtualization". *In Proc. of the nineteenth ACM SOSP*, 2003.

[3] S. Graupner, V. Kotov, and H. Trinks, "Resource-Sharing and Service Deployment in Virtual Data Centers".,*In Proc. of the 22nd ICDCS*, pp 666-674, July, 2002.

[4] VMware, Inc. VMware ESX Server User's Manual Version 1.5, Palo Alto, CA, April 2002.

[5] M. Reiser and S. S. Lavenberg, "Mean-Value Analysis of Closed Multichain Queueing Networks," J. ACM, vol. 27, pp. 313-322, 1980.

[6] A. Seidmann, P. J. Schweitzer, and S. Shalev-Oren, "Computerized Closed Queueing Network Models of Flexible Manufacturing Systems," Large Scale Systems, vol. 12, pp. 91-107, 1987.

[7] D. Menasce and V. Almeida, Capacity Planning for Web Services: Metrics, Models, and Methods: Prentice Hall PTR, 2001.

[8] A. Chandra, W. Gong, and P. Shenoy. "Dynamic Resource Allocation for Shared Data Centers Using Online Measurements". In *Proc. of International Workshop on Quality of Service*, June 2003.

[9] R. Doyle, J. Chase, O. Asad, W. Jin, and A. Vahdat. "Model-Based Resource Provisioning in a Web Service Utility". In *Proc. of the 4th USENIX USITS*, Mar. 2003.

[10] R. Levy, J. Nagarajarao, G. Pacifici, M. Spreitzer, A. Tantawi, and A. Yousse, "Performance Management for Cluster Based Web Services". in *Proc. of IFIP/IEEE 8th IM*, 2003.

[11] L. Slothouber. "A Model of Web Server Performance". In *Proc. of* Int'l World Wide Web Conference, 1996.

[12] B. Urgaonkar and P. Shenoy. Cataclysm. "Handling Extreme Overloads in Internet Services". In *Proc. of ACM SIGACT-SIGOPS PODC,* July 2004.

[13] T. Kelley. "Detecting Performance Anomalies in Global Applications". *In Proc.of Second USENIX Workshop on Real, Large Distributed Systems (WORLDS 2005)*, 2005.

[14] E. Cecchet, A. Chanda, S. Elnikety, J. Marguerite and W. Zwaenepoel. "A Comparison of Software Architectures for E-business Applications". *In Proc. of 4th Middleware Conference*, Rio de Janeiro, Brazil, June, 2003.

[15] Y. Udupi, A. Sahai and S. Singhal, "A Classification-Based Approach to Policy Refinement". *In Proc. of The Tenth IFIP/IEEE IM*, May 2007. (to appear).

[16] B. Urgaonkar, P. Shenoy, A. Chandra, and OP. Goyal. "Dynamic Provisioning of Multi-tier Internet Applications". In *Proc. of IEEE ICAC*, June 2005.

[17] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi, "An Analytical Model for Multi-tier Internet Services and its Applications". In *Proc. of ACM SIGMETRICS*, June 2005.

[18] C. Stewart, and K. Shen. "Performance modleing and system management for multi-component online services". In Proc. of USENIX NSDI, 2005.

[19] X. Liu, J. Heo, and L. Sha, "Modeling 3-Tiered Web Applications". In Proc. of 13th IEEE MASCOTS, Atlanta, Georgia, 2005.

[20] TPC Council, "TPC-W", http://www.tpc.org/tpcw.

[21] A. Zhang, P. Santos, D. Beyer, and H. Tang. "Optimal Server Resource Allocation Using an Open Queueing Network Model of Response Time". HP Labs Technical Report, HPL-2002-301.