Autonomic Load-Testing Framework

Cornel Barna
Department of Computer Science and Engineering
York University
Toronto, Canada
cornel@cse.yorku.ca

Marin Litoiu
Department of Computer Science and Engineering
York University
Toronto, Canada
mlitoiu@yorku.ca

Hamoun Ghanbari
Department of Computer Science and Engineering
York University
Toronto, Canada
hamoun@cse.yorku.ca

ABSTRACT

In this paper, we present a method for performance testing of transactional systems. The methods models the system under test, finds the software and hardware bottlenecks and generate the workloads that saturate them. The framework is autonomic, the model and workloads are determined during the performance test execution by measuring the system performance, fitting a performance model and by analytically computing the number and mix of users that will saturate the bottlenecks.

We model the software system using a two-layer queuing model and use analytical techniques to find the workload mixes that change the bottlenecks in the system. Those workload mixes become stress vectors and initial starting points for the stress test cases. The rest of test cases are generated based on a feedback loop that drives the software system towards the worst case behaviour.

Categories and Subject Descriptors

D.4.8 [**Software Engineering**]: Performance—*modeling and prediction, queueing theory*

General Terms

Performance

Keywords

performance testing, autonomic system, performance models, stress testing

1. INTRODUCTION

Performance testing is fundamental in assessing the performance of software components as well as of an entire software system. A major goal of performance testing is to uncover functional and performance problems under load and the root cause of those problems. Functional problems are often bugs, deadlocks and memory management bugs, buffer overflows. Performance problems often refer to high response time or low throughput under load.

In practice, the testing is done under operational conditions, that is, the testing is typically based on the expected usage of the system once is deployed and on the expected workload. The workload consists of the types of usage scenarios and the rate of these scenarios. A performance test usually lasts for several hours or even a few days and only tests a limited number of workloads. The major drawback of this approach is that expected usage and scenario rates are hard to predict. As a result, many workloads that the system will face remain uncovered by the stress test.

In this paper we propose an autonomic framework that explores the workload space and searches for points in this space that cause the worst case behaviour for software and hardware components of the system. It is generally known that the performance of a software system is influenced by the *hardware* and *software bottlenecks*. Bottlenecks are those resources where the requests are queued and delayed because the processing capacity limits of that resource. When those limits are reached, we say that the bottlenecks are *saturated*. Consider for example a web based application in which a web server has 100 threads available. When there are more than 100 pending requests, the server is a saturated bottleneck because it has reached its capacity. If the requests keep coming, they will be buffered in a waiting queue that will eventually reach its limits as well. In a software system, there are many bottlenecks and, more importantly, those bottlenecks change as the workload changes.

Finding the workloads that cause the bottlenecks to change is a challenging but rewarding problem. We propose an autonomic load stress testing framework that drives the workloads towards the points that create the bottlenecks and eventually saturate them. We also show that the software performance metrics reach their maximum or minimum for those workloads that cause some bottlenecks to reach their capacity or policy limits. The method uses an analytical representation of the software system, a two-layer queuing model that captures the hardware and software contention for resources. The model is automatically tuned, using online estimators that find the model parameters. The overall testing method is autonomic, based on a feedback loop that *generates workloads* according to the outputs of the model, *monitors* the software system under test, extract metrics, *analyzes* the effects of each workload and *plans* the new workloads based on the results of the analysis.

The type of software systems that would benefit the most from the proposed method are web based transactional sys-

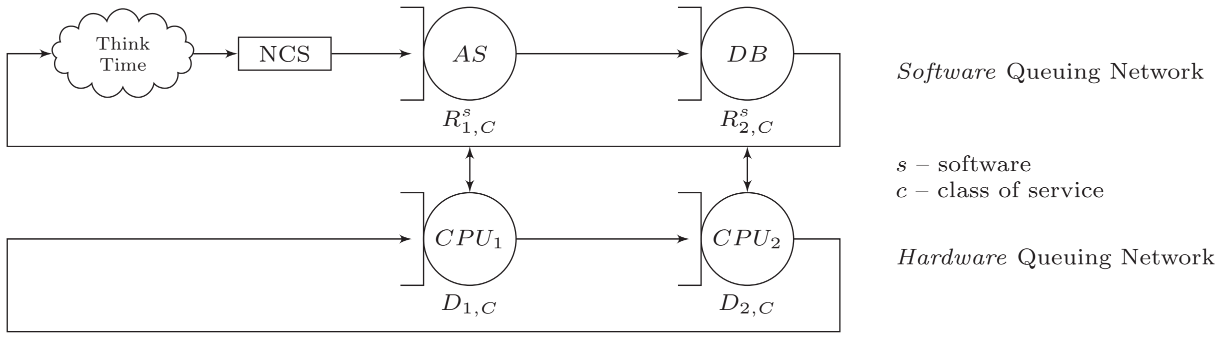


Figure 1: Software and hardware layers in a two tier web system

tems. To model the interaction of users with such systems, we define *classes of services*, or *classes* in short. A *class* is a service or a group of services that have similar statistical behavior and have similar requirements. When a user begins interacting with a service, a *user session* is created. The session will be maintained active until the user logs out or when he is inactive for a specified period of time. If we define N as the number of active users at some moment t, these users can use different classes of services. If we have C classes and N_c is the number of users in class C, then $N = N_1 + N_2 + \cdots + N_C$. N is also called *workload intensity* or *population* while combinations of N_c are called *workload mixes* or *population mixes*.

The remainder of the paper is organized as follows. Section 2 presents the theoretical foundations of the testing method. Section 3 describes the general testing framework and introduces the testing algorithm. A case study is presented in Section 4 and related work is described in Section 5. Conclusions and further work are presented in Section 6.

2. PERFORMANCE STRESS SPACE

This section introduces the *stress space*, defined as the multidimensional domain that can be covered by the software performance metrics. To start with, a software-hardware system can be described by two layers of queuing networks [30, 23]. The first layer models the software resource contention, and the second layer models the hardware contention.

To illustrate the idea, consider a web based system with two tiers, an application software server (AS) and database software (DB) server (see Figure 1). Each server runs on its dedicated hardware, CPU_1 and CPU_2 computers respectively. Consider also that we have two classes of service. The hardware layer can be seen as a queuing network with two queues (for simplicity of presentation, we only consider the CPUs of the servers in this example) and with the *demands* (or *service times*) for class C being $D_{1,C}$ and $D_{2,C}$ respectively, $C \in \{1,2\}$. The software layer has two queuing centres, the processes AS and DB, which queue requests whenever there are no threads available. (Besides queuing for threads, the requests can queue for *critical sections*, *semaphores*, etc. Those can be represented as queuing centres as well.)

The software layer also has non-critical sections (NCS) where there are no queuing delays and a Think Time centre that models the user think time between requests. The service times (demands) at the software layer are the *response times* of the hardware layers. In our case they are $R^s_{1,c}$ and $R^s_{2,c}$, and they include the demand and the waiting time at the hardware layer (we use the upper script s to denote software metrics that belong to the software layer).

2.1 Utilization Constraints

In multiuser, transactional systems, a hardware bottleneck is a device (CPU, disk, network) that has the potential to saturate if enough users access the system. In general, the device with the highest demand is the bottleneck. However, when there are many classes of requests with different demands at each device, then the situation becomes more complex. When the workload mix changes, the bottleneck in the system can change as well and there may be many simultaneous bottlenecks in the system at a given time.

To find all the hardware bottlenecks in the system, let's assume that the workload intensity is high enough to make the potential bottlenecks saturate. Workload mixes yield per class utilization at each resource; the sum of per class utilizations equals the total utilization of that resource. Total utilization of resource K_i is a linear function of per class utilizations and has to be less that physical capacity or policy constraints [19]:

$$U_K = \sum_{\forall C \in \mathcal{C}} \frac{D_{K,C}}{D_{K_r,C}} U_{K_r,C} < b_K, \quad \forall K \in \mathcal{K} \tag{1}$$

or exact the physical capacity or policy constraints:

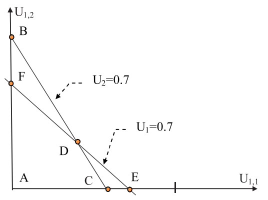
$$U_K = \sum_{\forall C \in \mathcal{C}} \frac{D_{K,C}}{D_{K_r,C}} U_{K_r,C} = b_K, \quad \forall K \in \mathcal{K} \tag{2}$$

where $0 \leq U_{K_r,C} \leq b_K$, $\forall C \in \mathcal{C}$; $\mathcal{C}$ and $\mathcal{K}$ are the sets of classes and resources; $K_r \in \mathcal{K}$ is a reference resource shared by all classes of request[1]; U_K is the total utilization of resource K; $U_{K_r,C}$ is the utilization of resource K_r by requests of class C, and $D_{K,C}$ is the demand of the resource K in class C. b_K is the utilization limit for resource K; for example, the maximum utilization of a single CPU device is 1, the utilization of a dual core CPU is 2, etc.

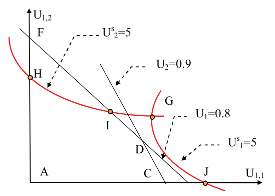
For the example described in Figure 1, if device 1 (CPU_1) is the shared device (K_r) and if we represent the inequation (1) in the reference device utilization space, we obtain the diagram of Figure 2a, where each segment represents one of the equations (2). The *stress space* is within the area AFDC. The coordinates $(U_{1,1}, U_{1,2})$ of the points F, D, C can be

[1]The existence of a resource shared by all classes simplifies the analysis; the results presented in the paper are valid without this assumption.

found by solving the system of equalities (2). The coordinates are those values for which one or more equations reach the limits b_K. On segment FD, device 1 is the bottleneck and on segment DC device 2 is the bottleneck. Note that we cannot drive the system out of the performance stress area because we either would exceed the capacity limits or violate policy constraints.



(a) *Hardware.*



(b) *Hardware and software.*

Figure 2: Constraints on the stress space.

2.2 Software Constraints

We can extend the above discussion for the software layer. Consider $1 \dots L$ software queues at the software layer.

Since the software layer is a normal queuing network (*separable* queuing network, a subset of general networks of queues, where assumptions like Flow Ballance Assumption hold [16]), we can apply the general queuing laws. Thus, using the utilization law [16], the utilization of a software resource L in class C is:

$$U^s_{L,C} = X^s_C \times R^s_{L,C}, \quad L \in \mathcal{L}, C \in \mathcal{C} \tag{3}$$

where $\mathcal{L}$ denotes the set of all software resources in the distributed system and $\mathcal{C}$ the set of all classes of services. The total utilization of a software resource L is:

$$U^s_L = \sum_{C \in \mathcal{C}} X^s_C \times R^s_{L,C} \tag{4}$$

Assuming that there exists a *hardware resource* K_r shared by all classes (for example a shared web server's CPU), we can express the utilization of resource K_r in class $C \in \mathcal{C}$:

$$U_{K_r,C} = X_C \times D_{K_r,C} \tag{5}$$

The throughput at the both hardware layers must be the same, at steady state both layers process the same number of requests/seconds, therefore $X_C = X^s_C$. By replacing X^s_C in (3) with the one from (5) and performing some simple algebraic operations, we can express the utilization of any software resource L in class C as a function of utilization of hardware resource K_r in the same class C:

$$U^s_{L,C} = U_{K_r,C} \frac{R^s_{L,C}}{D_{K_r,C}}, \quad \forall L \in \mathcal{L}, \forall C \in \mathcal{C} \tag{6}$$

Thus, using resource K_r as reference, we can rewrite (4) as:

$$U^s_L = \sum_{C \in \mathcal{C}} U_{K_r,C} \frac{R^s_{L,C}}{D_{K_r,C}}. \tag{7}$$

The utilization of each software contention centre L is limited by the capacity or policy constraints b_L, and that can be expressed as:

$$U^s_L = \sum_{C \in \mathcal{C}} U_{K_r,C} \frac{R^s_{L,C}}{D_{K_r,C}} < b_L, \quad \forall L \in \mathcal{L} \tag{8}$$

and equation (2) can be rewritten as:

$$U^s_L = \sum_{C \in \mathcal{C}} U_{K_r,C} \frac{R^s_{L,C}}{D_{K_r,C}} = b_L, \quad \forall L \in \mathcal{L} \tag{9}$$

where $0 \leq U^s_{L,C} \leq b_L$, $\forall C \in \mathcal{C}$.

These equations are non-linear because the terms $R^s_{L,C}$ depend non-linearly on $U_{K_r,C}$ [29], i.e $R^s_{L,C} = h(U_{K_r,C})$, where h is a non-linear function. The function h is the queuing network at the hardware layer. $R^s_{L,C}$ can be computed by solving the hardware queuing network model.

For the example described in Figure 1, if device 1 (CPU_1) is the shared device K_r and if we represent the equation (9) in the reference device utilization space, then we obtain the diagram of Figure 2b. The *stress space* is within the area AHIDC and is certainly different than when we consider only hardware resources. The coordinates $(U_{1,1}, U_{1,2})$ of the points H, I, G, J and the corresponding segments can be found by solving the equation 3. The coordinates are those values for which one or more equations reach the limits b_L. Note that some of the points are outside of the stress area, they cannot be reached. By taking into account the software constraints, the bottlenecks will evolve as follows: on segment HI, software entity AS is the bottleneck, on segment ID the hardware device 1 is the bottleneck and on segment DC device 2 is the bottleneck. Note that we cannot drive the system out of the stress area because either we would exceed the capacity limits or we violate policy constraints. Therefore, software entity DB is never saturated, although it comes very close.

In mathematical programming terms ([3]), the points B, C, D, E, F, H, I, G, J in Figures 2a and 2b are called *extreme points*. *Extreme points* are those points in the solution space where $U_K = b_K$ or $U^s_L = b_L$, for some hardware or software resource K or L. The domain delimited by the most interior constrains (like AHIDC in Figure 2a) is our *feasible stress space*. In mathematical programming terms, a linear function will reach the maximum or minimum in the extreme points of its feasible space. A non-linear function will reach its extreme values on the boundary of the feasible space. Therefore, the maximum (or minimum) of any performance stress metric (response time, throughput, buffer length, utilization, number of threads, etc.) is achieved on the boundary of the feasible stress space. It turns out that if we can explore the boundary, then we can find the maximum or the minimum of those metrics.

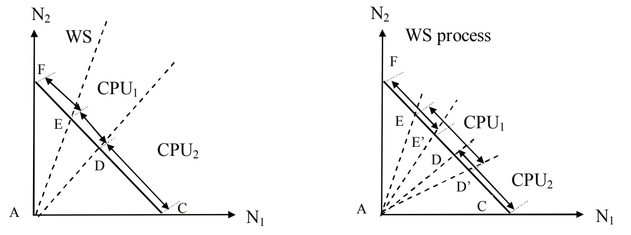
2.3 Workload Stress Vectors

Since we know how to analytically compute the feasible stress space boundary, including the *extreme points*, we need a mechanism to reach those boundaries on the real system. Unfortunately, we cannot drive utilization directly. On the real system we stress the system by generating the workloads, i.e. by accessing the URLs and by synthetically generating a number of users for each request type.

Note that neither N nor N_C are directly visible in (1) and (8), but they are directly involved in producing per class utilizations $U_{K_r,c}$. Our hypothesis is that it is possible to find N_C, when N is known, by using the solutions of equations (2) and (9). We rely on an early result, established for the asymptotic case for one layer hardware queuing networks in [2]. Our conjecture is that, if a solution of equation (2) and/or (9) is $U^*_{K_r,C}$, then the workload mix that yields that solution can be approximated as:

$$\beta_i^* = \frac{N_C}{N} = U_{K_r,C}^*. \tag{10}$$

The vectors $\beta^* = \langle \beta_1^*, \ldots, \beta_{|\mathcal{C}|}^* \rangle$ are *workload stress vectors* and are found by solving the equations (2) and (9) and computing all per class utilizations using (6). Figure 3 shows the stress vectors in the space of N_1 and N_2 (dashed lines), the number of users in class 1 and 2 respectively. On the dashed lines the ratio of users remains constant. When the software and hardware entities do not saturate, there is one bottleneck on each sector (a). When the entities saturate, then we can have multiple saturation devices for a range of population mixes. For example, both WS server process and the CPU of Application Server are bottlenecks on the segment EE' in Figure 3.



(a) *Non-saturated bottlenecks.* **(b)** *Saturated bottlenecks.*

Figure 3: Bottlenecks in population mix space, $N = N_1 + N_2$. As population mix changes, the bottleneck shifts.

3. THE AUTONOMIC TESTING FRAMEWORK

Figure 4 shows the proposed framework for autonomic performance testing. The framework will drive the system along the workload stress vectors until a performance stress goal is reached. A *stress goal* is target performance metric threshold, such as a software or hardware utilization value, a target response time or throughput for a class of request, etc.

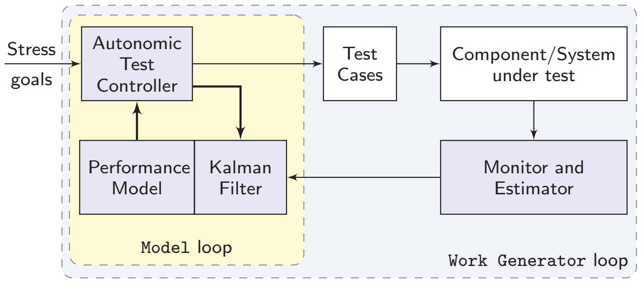


Figure 4: Autonomic performance stress testing.

An *autonomic test controller* runs the performance stress algorithm that will be presented later. In a nutshell, at each iteration it simulates a number of users that simultaneously access the system/component that is tested. Based on current state of the system and on the *stress goals*, a new workload will be computed and generated. Basically, the algorithm drives the system along the feasible stress test boundary or along the stress vectors.

During the test, the system is continuously monitored by a *performance monitor* and performance data is extracted. Data includes *CPU utilization*, *CPU time*, *disk utilization*, *disk time*, *waiting time* (which includes time waiting in critical sections, thread pools, connection pools), *throughput*, etc. Also, the *monitor* component will extract information about the workload that generated the data and information about the system. The monitored data is filtered through an estimator for error correction and noise removal. Estimators, like Kalman filters [14], have been proven to be very effective in estimating the demands [7] and we have used those in our implementation.

The performance data is passed to the *performance model* made of two queuing network layers. The model has 3 main functions: (a) it computes the $R_{K,C}$ (see Figure 1); (b) provides the equations and solutions for the load stress vectors (10) and (c) is used by the workload generator in lieu of real system to navigate along the stress vectors.

3.1 The Autonomic Test Controller

This section presents in detail the algorithm run by the autonomic test controller. After the classes of service are defined, the framework will use a model to make estimations about the number of users required and the classes they should execute in order to reach the stress goal (for example, utilization or response time above a specified threshold).

By solving equations (2) and (9) and then using (10) we can compute the *workload stress vector* β. Now the goal becomes finding the total number of users N that will drive the system on the feasible space boundary along the stress vectors.

We have developed the *Stress Algorithm* that will find the number of users and their mix that will first reach the stress goal. That will guarantee that beyond that number, we either go beyond the policy constraints (when they limit the feasible space) or we are guaranteed we stay on the boundary of feasible space.

After solving equations (2) and (9) and getting the system bottlenecks, the algorithm has two loops:

- in the first loop (`Model` *loop* in Figure 4) the number of users to reach the boundary of the feasible space on each stress vector and saturate a bottleneck is computed on the model;
- in the second loop (`Work Generator` *loop* in Figure 4) the algorithm works with the real system, submitting requests and measuring the performance. This loop is initialized with the values from the first loop and corrects the eventual errors inherent in working with a model instead of the real system.

Both loops follow similar feedback ideas: having an *extreme point p* and the total number of users N, the *workload stress vector* is computed by using relations (10); then the performance metric corresponding to the stress goal for this workload mix is determined, either by solving the model (`Model` *loop*), or by generating workload and measuring the performance metric on the real system (`Work Generator` *loop*). This performance metric is compared with the target value. If the stopping condition is not met, the framework will use a hill-climbing strategy to find a new value for N and a new iteration will start. In our algorithm we stop each loop when the predicted/measured performance metric is within *err*% from the target value. However, other con-

Algorithm 1: Stress Algorithm – algorithm to find the number of users that will bring the performance metric of a resource $K \in \mathcal{K} \cup \mathcal{L}$ at a target value PM_K.

```
input: $N$ – the initial number of users
input: $PM_K$ – the targeted performance metric for resource
       $K \in \mathcal{K} \cup \mathcal{L}$
input: $err$ – accepted error
1  Tune the model by measuring and adjusting the service
   demands for each class;
2  Find all extreme points by solving the equations (2) and (9);
3  Compute the workload stress vectors, by using (10);
4  foreach stress vector $p \in \mathcal{P}$ do
5      $pm_{e,K} \leftarrow -1$; // estimated performance metric
6      $pm_{m,K} \leftarrow -1$; // measured performance metric
7      Tune the model for stress vector $p$ and $N$ users;
       // Stop when the estimated performance metric is
          within $err\%$ from the target performance metric
8      while $\left|1 - \frac{pm_{e,K}}{PM_K}\right| > err$ do
9          Compute $\langle N_1, N_2, \ldots, N_{|\mathcal{C}|}\rangle$ for $N$ and $p$;
10         Solve model for $\langle N_1, N_2, \ldots, N_{|\mathcal{C}|}\rangle$;
11         Update $pm_{e,K}$ with the estimated value;
12         if $\left|1 - \frac{pm_{e,K}}{PM_K}\right| > err$ then
13             Update $N$ using a hill climbing strategy;
14     Generate workload and measure the metrics;
15     Update $pm_{m,K}$ with the value measured;
16     if $\left|1 - \frac{pm_{e,K}}{pm_{m,K}}\right| > err$ then
17         go to line 7;
18     while $\left|1 - \frac{pm_{m,K}}{PM_K}\right| > err$ do
19         Compute $\langle N_1, N_2, \ldots, N_{|\mathcal{C}|}\rangle$ for $N$ and $p$;
20         Generate workload and measure the metrics;
21         Update $pm_{m,K}$ with the value measured;
22         if $\left|1 - \frac{pm_{m,K}}{PM_K}\right| > err$ then
23             Update $N$ using a hill climbing strategy;
```

ditions can be used, such as the performance metric is with at most $err\%$ *above* the target or a combination of multiple resources' metrics.

In the first step (line 1) the algorithm tunes the two-layer model. In essence it estimates the demands $D_{K,C}$ for hardware resources. The demands are important for the system of equations (2) and (9). For most infrastructures, per class service times or demands are hard or costly to measure. We estimate those values by using Kalman filter as illustrated by the Model Tuning Algorithm.

In order to get the demands for a class, we generate workload by considering that all users will be in that class and no user will access other classes (line 2). Then, for each resource we execute a loop to find the correct value for demand: we solve the model to extract the estimated value and also measure the performance metric—if the two values are close enough (again, our criterion is that the estimated value is within $err\%$ from the measured value, but other criteria can be used) then we accept the demand found by the Kalman filter, else we move to the next iteration.

4. EXPERIMENTS

We tested our framework on a a web application deployed over cluster with three Windows XP machines: one Database Server (MySQL) and two Web Servers (Tomcat). Also we

Algorithm 2: Model Tuning Algorithm – estimate the demands for each resources in each class.

```
input  : $N$ – the number of users
input  : $err$ – the accepted error
output: $D$ – demands matrix, of size $|\mathcal{C}| \times |\mathcal{K} \cup \mathcal{L}|$
1  for $i \leftarrow 1$ to $|\mathcal{C}|$ do
2      $\mathbb{N} \leftarrow \langle 0, 0, \ldots, 0\rangle$;
3      $N_i \leftarrow N$;
4      Generate workload;
5      foreach $K \in \mathcal{K} \cup \mathcal{L}$ do
6          while $\left|1 - \frac{pm_{e,K}}{pm_{m,K}}\right| > err$ do
7              Solve model;
8              Update $pm_{e,K}$ with the model estimated value;
9              Update $pm_{m,K}$ with the measured value;
10             if $\left|1 - \frac{pm_{e,K}}{pm_{m,K}}\right| > err$ then
11                 Estimate service demands using Kalman
                   filters;
12                 Update model with the estimated service
                   demands;
13         Update $D_{C_i,K}$ with the last value estimated by
           Kalman filters;
```

had on a machine a Workload Balancer (Apache) to distribute the incoming web requests to the two web servers. Figure 5 shows our deployed testing framework that is a materialization of the logical structure presented earlier in Figure 4.

On each machine we had installed monitoring tools to be able to measure the performance metrics: Windows XP SNMP agents, JMX and Windows Performance Counters. The workload generator and the analysis of the performance data was on a separate machine.

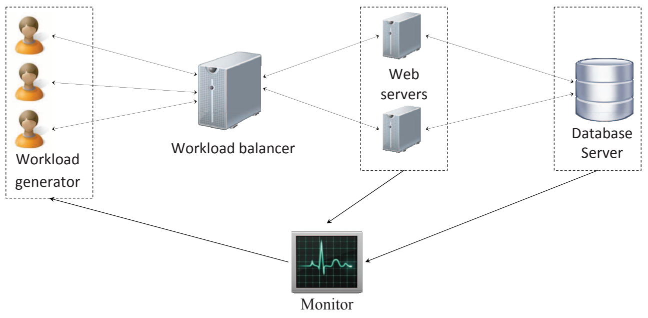


Figure 5: The cluster used for experiments.

On the cluster we have installed a typical 3-tier application–an online store–with 3 main scenarios:

- `browse` – the user is browsing through the available items in the store. Also the user will be able to specify how many items he wants to have in a single page (which is a parameter for the scenario);
- `buy` – the user decides to buy some items and add them in the shopping cart;
- `pay` – the user goes to checkout and pays for the content of the shopping cart;

The two-layer model of the application is the one represented in Figure 1, earlier in the paper. At the software layer, we have the two software queuing centres, the Web

Servers and the Database. The load balancer is not represented in this particular example as a queuing centre because it is performant enough not to queue requests (however, in a general case it should be represented). Therefore, $\mathcal{L} = \langle Web, Data \rangle$.

The hardware layer is made of two queuing centres $\mathcal{K} = \langle CPU_{web}, CPU_{data} \rangle$.

The application was modeled with Apera tool [1], developed by one of the authors.

Initially, we have 3 classes of requests, represented by the 3 scenarios, $\mathcal{C} = \{\texttt{browse}, \texttt{buy}, \texttt{pay}\}$.

4.1 Classes of Service

In general, the number of classes is equal with the number of scenario because, most of the times, the performance metrics are not significantly influenced by the arguments values and we can consider that a scenario is a single class of service. Alternatively, we can consider each scenario with the maximum argument for stress testing or average argument for performance testing. However, there are situations when we need to split a scenario in more classes. In our testing framework, we probe each scenario with randomly generated arguments and we measure the resulting stress goal metric, for example CPU utilization[2] is measured. The scenarios that have a high variance in the performance metrics (utilization in our example) are most likely to provide significant improvements if they are split. The other scenarios will generate a single class.

When enough samples have been gathered, we can split the utilization interval in subintervals and for each such subinterval we will determine the corresponding range for scenario parameters. Each such range will generate a class of service.

A user executing the scenario `browse` will send a request to a web server, that will generate a request to a database server to select a number of items from database. The result will be sent back to the web server that will create a web page containing all the selected items. The number of items to be displayed will be specified as a parameter to the scenario.

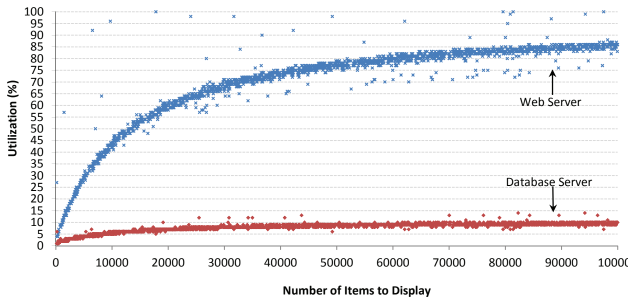


Figure 6: The CPU utilization on the web server and database server when the `browse` scenario is executed.

Figure 6 shows the measured CPU utilization on the web server and database server when the parameter goes from 0 to 100,000. We see that the CPU utilization at the web server increases fast at first (for values lower than 20,000) and then slows down.

[2]Any other performance metric can be used: throughput, disk utilization, etc.

The CPU utilization at the database server follows a similar pattern, although the maximum value is lower than 15%.

Because the web server CPU utilization grows faster and get closer to 100% we will use it to split the scenario into four classes (first class for utilization between 0 and 25%, second class for utilization between 25% and 50%, etc.). For that, considering that the utilization follows a logarithmic shape, we can do regression. Table 1 summarizes the ranges for the parameter corresponding to the four classes of service.

Scenario	Range		
`browse 0`	0	-	2,855
`browse 1`	2,856	-	11,752
`browse 2`	11,753	-	48,370
`browse 3`	48,371	-	100,000

Table 1: The ranges of the parameter when the scenario is split in four classes.

Finding the right number of classes to split a scenario is a hard problem on it's own. If we have too many classes, the accuracy of the algorithm will increase, but so does the complexity. Too few classes, and the model will be solved very fast at the expense of precision.

4.2 Results

Once we decided the number of classes of services, we run the testing algorithm. At the first step it estimates the demands for each class of service using the Model Tuning Algorithm (Table 2 shows the values found when the model is calibrated for CPU utilization).

	CPU_{web}	CPU_{data}
`buy`	5.38	5.60
`pay`	5.17	5.60
`browse 0`	41.32	11.78
`browse 1`	192.84	39.91
`browse 2`	769.29	153.20
`browse 3`	1,961.82	372.30

Table 2: The values for demands for each scenario found using Kalman filters (milliseconds).

We run the framework for 3 stress goals: (a) hardware utilization, (b) web container number threads and (c) response time for each scenario.

The hardware utilization stress goal. This goal aims at performance of the system when a hardware resource runs at a threshold utilization. This can be maximum utilization 100%, if reachable, or less than that (to resemble the operational conditions). When the system reaches the target conditions, performance metrics are collected and analyzed. For illustration purposes, we set the target utilization at 50% and want to find the number of users for each stress vector that will yield that utilization.

The stress algorithm found 22 bottlenecks in the system (the workload stress vectors are shown in Table 3). Each one is a vector of six values, each value representing the proportion of the users that have to access that scenario in order to saturate the bottleneck. The order of the scenarios considered is $\langle$`buy`, `pay`, `browse 0`, `browse 1`, `browse 2`, `browse 3`$\rangle$.

#	Workload stress vectors					
1	⟨ 1,	0,	0,	0,	0,	0 ⟩
2	⟨ 0,	1,	0,	0,	0,	0 ⟩
3	⟨ 0,	0,	1,	0,	0,	0 ⟩
4	⟨ 0,	0,	0,	1,	0,	0 ⟩
5	⟨ 0,	0,	0,	0,	1,	0 ⟩
6	⟨ 0,	0,	0,	0,	0,	1 ⟩
7	⟨ 0.947,	0,	0.053,	0,	0,	0 ⟩
8	⟨ 0.985,	0,	0.015,	0,	0,	0 ⟩
9	⟨ 0.952,	0,	0,	0.048,	0,	0 ⟩
10	⟨ 0.990,	0,	0,	0.010,	0,	0 ⟩
11	⟨ 0.952,	0,	0,	0,	0.048,	0 ⟩
12	⟨ 0.991,	0,	0,	0,	0.009,	0 ⟩
13	⟨ 0.953,	0,	0,	0,	0,	0.047 ⟩
14	⟨ 0.991,	0,	0,	0,	0,	0.009 ⟩
15	⟨ 0,	0.896,	0.104,	0,	0,	0 ⟩
16	⟨ 0,	0.970,	0.030,	0,	0,	0 ⟩
17	⟨ 0,	0.905,	0,	0.095,	0,	0 ⟩
18	⟨ 0,	0.980,	0,	0.020,	0,	0 ⟩
19	⟨ 0,	0.906,	0,	0,	0.094,	0 ⟩
20	⟨ 0,	0.981,	0,	0,	0.019,	0 ⟩
21	⟨ 0,	0.907,	0,	0,	0,	0.093 ⟩
22	⟨ 0,	0.982,	0,	0,	0,	0.018 ⟩

Table 3: The workload stress vectors found.

Str. vec.	Mx.	Usr (e)	Usr (m)
1	3	281	282
2	10	265	275
3	4	70	72
4	2	16	16
5	4	5	5
6	6	3	3
7	9	236	252
8	10	270	279
9	4	211	213
10	5	264	266
11	7	98	96
12	4	244	245
13	9	59	54
14	3	190	190
15	3	248	249
16	3	257	258
17	6	138	142
18	6	236	238
19	6	53	53
20	12	196	180
21	6	29	27
22	8	136	142

(a) *Only hardware queues.*

Str. vec.	Mx.	Usr (e)	Usr (m)
1	6	258	261
2	4	249	248
3	7	62	70
4	2	16	17
5	2	6	6
6	3	3	3
7	4	237	239
8	4	254	256
9	3	204	204
10	5	245	247
11	2	88	88
12	5	239	239
13	2	54	54
14	4	180	179
15	3	232	233
16	4	242	244
17	8	136	131
18	9	227	234
19	5	42	42
20	4	188	187
21	6	30	31
22	8	147	142

(b) *Hardware and software queues.*

Table 4: The users number that will bring the CPU utilization (on web server or database server) above 50%.

For each stress vector, the first loop of the stress algorithm uses the model to find the number of users N that will bring the utilization at the specified threshold (50% in our experiments). Then the workload generator will simulate N users and measure the metrics. If the measured and the estimated metric values are not close (in our experiments, within 10% from each other) a new calibration of the model is performed and then the first loop is executed again. If the two values are close, N is finely tuned by the second loop which sends requests to the real system.

From Table 4, we can see that the number of users computed by the first loop (column `Usr (e)`) using only the model was very close to the actual number found by the second loop (column `Usr (m)`). Therefore very few workloads were used against the real system (column `Mx.`). For this particular example the hill climbing method of the second loop tried less than 10 workloads for each stress vector, with the exception of the 20^{th} vector.

When we limit the number of threads on the web server to 5 (thus creating software queues), we see that most of the times the number of users required to bring the CPU utilization above the 50% threshold is less, as shown in Table 4b.

What we can infer from Table 4 is that the 50% utilization can be reached for a variety of workloads. For example, in Table 4a, if we go along the stress vector 1, the utilization is reached for 282 users. Along the vector 6, the same utilization is reached for 3 users. This certainly shows a very big limitation of the system when the workloads are shifted toward the class 6. Similar conclusions can be drawn for the case illustrated in Table 4b.

Software utilization goal. We run the algorithm again, targeting the software utilization of the web server. We set the maximum number of threads to handle requests on the web server to 5. The stress goal for the algorithm is 50% utilization of the container, meaning the use of at most 2.5 threads on average. When calibrated for this goal, the model found only 6 stress vectors, each one having all users executing a single scenario. The results are summarized in Table 5a.

We notice that the numbers predicted by the model (first loop of the algorithm) was not as accurate as in the hardware utilization case. Nevertheless, the second loop was were able to find the correct number (column `Usr (m)`). Because the model had to be re-calibrated several times, the number of generated workloads (column `Mx.`) is higher for some stress vectors. However, the total number of workloads generated was 43, which is significantly lower than trying all possible workloads.

Similar to hardware utilization, there is a large range of workloads that use 50% of the threads. This shows the pitfalls of not exploring the whole space, the web container can saturate with very few users, if they come in class 6.

The response time goal. In the last experiment, we wanted to see the maximum number of users that will ensure that the response time for requests on each scenario does not exceed a certain value. Because the demands for classes vary greatly, we chose a different target value for each class. Our threshold vector is ⟨50, 50, 100, 500, 1500, 5000⟩ for the classes ⟨`buy`, `pay`, `browse 0`, `browse 1`, `browse 2`, `browse 3`⟩ (the times are expressed in milliseconds). Tuning the model for this goal gave us 16 workload stress vectors. The results are shown in Table 5b.

Although the model was not as accurate as for hardware utilization, being necessary several re-tunnes, each stress vector required at most 15 workloads to be investigated (column `Mx.`) before the number of users is found (column `Usr (m)`). Usually the more accurate is the model, fewer workloads need to be generated.

We also noticed that for the stress vector 10 (which is ⟨0,

Str. vec.	Mx.	Usr (e)	Usr (m)
1	12	316	144
2	13	196	144
3	11	52	69
4	3	20	22
5	2	10	10
6	2	7	8

(a) *Web container utilization.*

Str. vec.	Mx.	Usr (e)	Usr (m)
1	9	185	171
2	13	233	170
3	9	98	93
4	4	28	27
5	2	9	9
6	2	8	8
7	15	149	119
8	15	111	87
9	15	20	107
10	2	3	3
11	3	36	36
12	3	45	45
13	7	17	14
14	4	31	33
15	11	2	9
16	5	30	30

(b) *Response times.*

Table 5: The number of users found when the performance metric is web container utilization and response time for each class.

0.224, 0.776, 0, 0, 0⟩—22.4% of the users execute scenario `pay` and 77.6% execute `browse 0`) there are necessary only 3 users to get a the response time greater than our threshold. If all users execute `pay` (stress vector 2) there are necessary 170 users, and it all of them execute `browse 0` (stress vector 3), then 93 users are required. The stress vector 10 shows the dramatic effect that the combination of the scenarios have.

Again, we notice that the total number of tested workloads, 119, is much lower than the size of the search space.

4.3 Complexity of the algorithm

The workload mix space is combinatorial in size. The number of total workload mixes when there are N users and C classes is given by the formula [18]:

$$Mixes_{N,C} = \binom{N+C-1}{C-1} \tag{11}$$

The number of classes is known from the begining, but the number of users is not, so we would have to consider that N goes from 0 to N_{max}, where N_{max} is the maximum value we allow for N. Thus the total number of mixes is:

$$Mixes = \sum_{N=1}^{N_{max}} \binom{N+C-1}{C-1} \tag{12}$$

Table 6 shows this combinatorial explossion:

N_{max}	C	$Mixes_{N_{max},C}$	Total Mixes
200	1	1	200
200	2	201	20 300
200	3	20 301	1 373 700
200	4	1 373 701	70 058 750
200	5	70 058 751	2 872 408 790
200	6	2 872 408 791	98 619 368 490

Table 6: The size of the workload mix space.

Exploring exhaustively the workload mix space is unfeasible. Our algorithm explores just a small fraction, without needing N_{max}, and finds the minimum number of users that will bring the targeted metric value above the specified treshold.

To better understand how the algorithm explores the workload mix space, lets consider only two classes: `buy` and `browse 0`. Figure 7 shows how the CPU Utilization (for web and database server), Web Container Utilization and Response Time (for each class) change with the workload mix. The utilizations are shown as percentages, and the response time is expressed in miliseconds. On each plot the threshold is shown as a plane parallel with the $X0Y$ plane, that intersects the Z-axis at 50 in Figures 7a, 7b, 7c and 7d and 100 in Figure 7e (the target was 50% utilization for CPU and web container, 50 ms response time for scenario `buy` and 100 ms response time for scenario `browse 0`).

The model found 4 bottlenecks and the stress vectors are shown with blue lines on the plots. The algorithm investigates only mixes on these lines. In the picture it can be seen that one of these lines will intersect the treshold plane in a point with minimum users among all other intersection points (it is always the vector that follows the Y-axis).

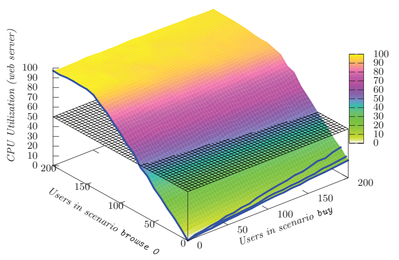
In our experiments, the framework was able to find the intersection point for one stress vector after less than 20 workload mixes were tried. That means our framework can find the overall minimum number of users with less than 80 workload mixes, which is a very low value compared with the size of the search space (20,300 for 2 clases and 200 maximum users).

5. RELATED WORK

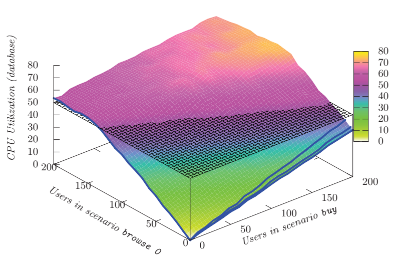
Transactional systems in the context of autonomic computing have been modeled by several authors as regression models or Queuing Network Models (QNM). Dynamic regression models have been described in [10, 21]. Queuing Network Models were described in [26] as predictive models.

Early work in finding bounds on response time and throughput for one dimension of the workloads (one class) was done in [35, 5, 16, 29]. In [2] the authors showed that in multiple workload mixes, multiple resources systems, changes in workload mixes can change the system bottleneck; the points in the workload mix space where the bottlenecks change are called *crossover points*, and the sub-spaces for which the set of bottlenecks does not change are called *saturation sectors*. The same authors, in the same paper, showed analytical relations between the workload mixes and utilization at the saturated bottlenecks as well as analytical expressions for asymptotic (with saturated resources) response times, throughput, and utilization within the saturation sectors. The results were presented for one queuing network layer consisting of hardware resources. In this paper, we considered two layers with the emphasis on the software layer. Moreover, our approach is defined in the context or performance testing.

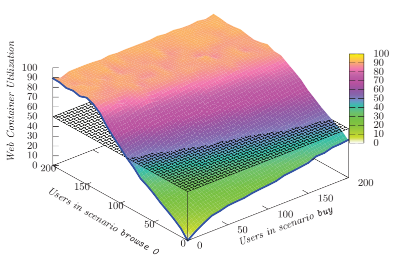
The paper [19] extended the results from [2] to non-asymptotic conditions (non-saturated resources), and used linear and non-linear programming methods for finding maximum object utilization across all workload mixes. That technique involved only the hardware bottlenecks. In our current approach, we consider the software bottlenecks and we combine the model search for worst case behaviour with a search on the real system.



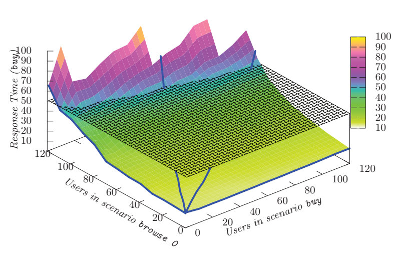
(a) *CPU utilization on web server.*



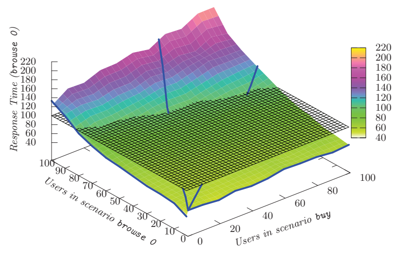
(b) *CPU utilization on database server.*



(c) *Web container utilization.*



(d) *Response time for* `buy`*.*



(e) *Response time for* `browse 0`*.*

Figure 7: On Z-axis is the CPU utilization on the servers (a and b), web container utilization (c) and the response time of the two classes of service (d and e) when there are N_1 users in the class `buy` (X-axis) and N_2 users in the class `browse 0` (Y-axis).

There is no fully automatic method for building the structure of a performance model, however, there are available tools that can help in building a structure of the performance model [1]. Recent papers, like [20, 34, 36], have shown how to build a tracking filter and a predictive QNM such that the model's outputs always match those of the real system. Performance parameters like the service time, think times, and the number of users can be accurately tracked and fed into a QNM. In our approach, we estimate the demands on the hardware layer using a method similar to [36]

To the best of our knowledge, there is no performance model driven testing approach similar the one presented in this paper. Although there are many model driven performance activities, they do not refer to testing. Many researchers have targeted capacity planning of distributed and client-server software systems and specially the web based ones [24, 4, 25]. Amongst those, many approaches have used the widely recognized queuing models to model web applications at operational equilibrium [24, 25, 9] which has resulted in automated building of measurement based performance models [33, 8] or capacity calculators [32]. Others have tried to model the effect of application and server tuning parameters on performance using statistical inference, hypothesis testing and ranking (e.g. [11, 31]). In a rather different approach some have tried to automate the detection of potential performance regressions, by applying statistics on regression testing repositories [6, 12, 13]. This had enabled developers to identify subsystems that show performance deviations in load tests [22].

All these approaches have contributed to designing scalable systems, building on-demand performance management systems [17, 27, 15] and performance aware software systems [28].

In this paper our focus was on model driven testing and on finding a method that drives the system towards a target state where performance metrics of interests can be collected. The model is fundamental in analysing the feasible stress space and in driving the system towards the saturation points.

6. CONCLUSIONS

This paper presented an autonomic performance testing method for stress testing web software systems. The systems are modeled with a two-layer Queuing Network Model. The model is used to find the software and hardware bottlenecks in the system and to give a hint about *workloads* that will saturate them. These hints are used as initial workloads on the real system and then in a feedback loop that guides the system towards a stress goal.

The workloads are characterized by *workload intensity*, which is the total number of users, and by the *workload mix*, which is ratio of users in each class of service. By extracting the *switching points* from the model, we are able to compute the *stress vectors* that yield a bottleneck change. Applying a hill-climbing strategy for workload intensity along the stress vectors, we are able to reach the stress goal.

We applied the method to find the workload intensity and workload mix that yields target software and hardware utilization limits or a target response time. The results show that the algorithm is capable to reach the target goal with a small number of iterations and therefore testcases.

Future work includes extending the framework to include more target goals, validate it on large scale software systems and address functional problems uncovered by stress testing.

7. REFERENCES

[1] Application Performance Evaluation and Resource Allocator (APERA), 2009. `http://www.alphaworks.ibm.com/tech/apera`.

[2] Gianfranco Balbo and Giuseppe Serazzi. Asymptotic analysis of multiclass closed queueing networks: multiple bottlenecks. *Performance Evaluation*, 30(3):115–152, 1997.

[3] Mokhtar S. Bazaraa, Hanif D. Sherali, and C. M. Shetty. *Nonlinear Programming: Theory and Algorithms.* Wiley-Interscience, second edition, 1993.

[4] Ágnes Bogárdi-Mészöly, Tihamér Levendovszky, and Ágnes Szeghegyi. Improved performance models of web-based software systems. In *INES'09: Proceedings of the IEEE 13th international conference on Intelligent Engineering Systems*, pages 23–28, Piscataway, NJ, USA, 2009. IEEE Press.

[5] Derek L. Eager and Kenneth C. Sevcik. Performance bound hierarchies for queueing networks. *ACM Trans. Comput. Syst.*, 1(2):99–115, 1983.

[6] K. C. Foo, Z. M. Jiang, B. Adams, A. E. Hassan, Y. Zou, P. Flora, and P. Engineering. Mining performance regression testing repositories for automated performance analysis.

[7] Hamoun Ghanbari, Cornel Barna, Marin Litoiu, Murray Woodside, Tao Zheng, Johnny Wong, and Gabriel Iszlai. Tracking adaptive performance models using dynamic clustering of user classes. In 2nd *ACM International Conference on Performance Engineering (ICPE 2011)*, New York, NY, USA, 2011. ACM.

[8] Hassan Gomaa and Daniel A. Menascé. Performance engineering of component-based distributed software systems. In *Performance Engineering, State of the Art and Current Trends*, pages 40–55, London, UK, 2001. Springer-Verlag.

[9] Neil J. Gunther. *Guerrilla Capacity Planning: A Tactical Approach to Planning for Highly Scalable Applications and Services.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[10] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems.* John Wiley & Sons, 2004.

[11] Gábor Imre, Tihamér Levendovszky, and Hassan Charaf. Modeling the effect of application server settings on the performance of j2ee web applications. In *TEAA'06: Proceedings of the 2nd international conference on Trends in enterprise application architecture*, pages 202–216, Berlin, Heidelberg, 2007. Springer-Verlag.

[12] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. Automated performance analysis of load tests. 2009.

[13] Zhen Ming Jiang, A.E. Hassan, G. Hamann, and P. Flora. Automatic identification of load testing problems. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 307–316, 2008.

[14] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering*, 82(Series D):35–45, 1960.

[15] A. Kraiss, F. Schoen, G. Weikum, and U. Deppisch. Towards response time guarantees for e-service middleware. *Bulletin of the Technical Committee on*, page 58, 2001.

[16] Edward D. Lazowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik. *Quantitative system performance: computer system analysis using queueing network models.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1984.

[17] R. Levy, J. Nagarajarao, G. Pacifici, A. Spreitzer, A. Tantawi, and A. Youssef. Performance management for cluster based web services. pages 247–261, March 2003.

[18] Marin Litoiu. A performance analysis method for autonomic computing systems. *ACM Transactions on Autonomous and Adaptive Systems*, 2(1):3, 2007.

[19] Marin Litoiu, Jerome Rolia, and Giuseppe Serazzi. Designing process replication and activation: A quantitative approach. *IEEE Trans. Softw. Eng.*, 26(12):1168–1178, 2000.

[20] Marin Litoiu, Murray Woodside, and Tao Zheng. Hierarchical model-based autonomic control of software systems. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, 2005.

[21] Ying Lu, Tarek Abdelzaher, Chenyang Lu, Lui Sha, and Xue Liu. Feedback control with queueing-theoretic prediction for relative delay guarantees in web servers. In *RTAS '03: Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, page 208, Washington, DC, USA, 2003. IEEE Computer Society.

[22] H. Malik, B. Adams, A. E. Hassan, P. Flora, and G. Hamann. Using load tests to automatically compare the subsystems of a large enterprise system.

[23] Daniel A. Menascé. Simple analytic modeling of software contention. *SIGMETRICS Performance Evaluation Review*, 29(4):24–30, 2002.

[24] Daniel A. Menascé and Virgílio A. F. Almeida. *Capacity planning for Web performance: metrics, models, and methods.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[25] Daniel A. Menascé and Virgílio A. F. Almeida. *Scaling for E Business: Technologies, Models, Performance, and Capacity Planning.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[26] Daniel A. Menascé and Mohamed N. Bennani. On the use of performance models to design self-managing computer systems. In *Proceedings of the Computer Measurement Group Conference*, pages 7–12, 2003.

[27] Daniel A. Menascé, Honglei Ruan, and Hassan Gomaa. Qos management in service-oriented architectures. *Perform. Eval.*, 64(7-8):646–663, 2007.

[28] Diego Perez-Palacin, José Merseguer, and Simona Bernardi. Performance aware open-world software in a 3-layer architecture. In *WOSP/SIPEW '10: Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*, pages 49–56, New York, NY, USA, 2010. ACM.

[29] M. Reiser and S. S. Lavenberg. Mean-value analysis of closed multichain queuing networks. *J. ACM*, 27(2):313–322, 1980.

[30] J. A. Rolia and K. C. Sevcik. The method of layers. *IEEE Transactions on Software Engineering*, 21(8):689–700, 1995.

[31] Monchai Sopitkamol and Daniel A. Menascé. A method for evaluating the impact of software configuration parameters on e-commerce sites. In *WOSP '05: Proceedings of the 5th international workshop on Software and performance*, pages 53–64, New York, NY, USA, 2005. ACM.

[32] Dharmesh Thakkar. Automated capacity planning and support for enterprise applications. Master's thesis, Queens University, 2009.

[33] Dharmesh Thakkar, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. A framework for measurement based performance modeling. In *WOSP '08: Proceedings of the 7th international workshop on Software and performance*, pages 55–66, New York, NY, USA, 2008. ACM.

[34] Murray Woodside, Tao Zheng, and Marin Litoiu. The use of optimal filters to track parameters of performance models. In *QEST '05: Proceedings of the Second International Conference on the Quantitative Evaluation of Systems*, page 74, Washington, DC, USA, 2005. IEEE Computer Society.

[35] J. Zahorjan, K. C. Sevcik, D. L. Eager, and B. I. Galler. Balanced job bound analysis of queueing networks. In *SIGMETRICS '81: Proceedings of the 1981 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, page 58, New York, NY, USA, 1981. ACM.

[36] Tao Zheng, Jinmei Yang, Murray Woodside, Marin Litoiu, and Gabriel Iszlai. Tracking time-varying parameters in software systems with extended kalman filters. In *CASCON '05: Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research*, pages 334–345. IBM Press, 2005.