A Cost-Sensitive Adaptation Engine for Server Consolidation of Multitier Applications

Gueyoung Jung[1], Kaustubh R. Joshi[2], Matti A. Hiltunen[2],
Richard D. Schlichting[2], and Calton Pu[1]

[1] College of Computing, Georgia Institute of Technology, Atlanta, GA, USA,
{gueyoung.jung,calton}@cc.gatech.edu
[2] AT&T Labs Research, 180 Park Ave, Florham Park, NJ, USA
{kaustubh,hiltunen,rick}@research.att.com

Abstract. Virtualization-based server consolidation requires runtime resource reconfiguration to ensure adequate application isolation and performance, especially for multitier services that have dynamic, rapidly changing workloads and responsiveness requirements. While virtualization makes reconfiguration easy, indiscriminate use of adaptations such as VM replication, VM migration, and capacity controls has performance implications. This paper demonstrates that ignoring these costs can have significant impacts on the ability to satisfy response-time-based SLAs, and proposes a solution in the form of a cost-sensitive adaptation engine that weighs the potential benefits of runtime reconfiguration decisions against their costs. Extensive experimental results based on live workload traces show that the technique is able to maximize SLA fulfillment under typical time-of-day workload variations as well as flash crowds, and that it exhibits significantly improved transient behavior compared to approaches that do not account for adaptation costs.

1 Introduction

Cloud computing services built around virtualization-based server consolidation are revolutionizing the computing landscape by making unprecedented levels of compute power cheaply available to millions of users. Today, platforms such as Amazon's EC2, AT&T's Synaptic Hosting, Google's App Engine, and Salesforce's Force.com host a variety of distributed applications including multitier enterprise services such as email, CRM, and e-commerce portals. The sharing of resources by such applications owned by multiple customers raises new resource allocation challenges such as ensuring responsiveness under dynamically changing workloads and isolating them from demand fluctuations in co-located virtual machines (VMs). However, despite the well-documented importance of responsiveness to end users [1, 2, 3], cloud services today typically only address availability guarantees and not response-time-based service level agreements (SLAs).

Virtualization techniques such as CPU capacity enforcement and VM migration have been proposed as ways to maintain performance [4, 5, 6, 7, 8, 9]. However, there is little work that considers the impact of the reconfiguration actions themselves on application performance except in very limited contexts. For

Before	Apache	% Chg.	Tomcat	% Chg.	MySQL	% Chg.
102.92	141.62	37.60	315.83	206.89	320.93	211.83

Table 1. End-to-End Response Time (ms) during VM Migration

example, while [10] shows that live migration of VMs can be performed with a few milliseconds of downtime and minimal performance degradation, the results are limited only to web servers. This can be very different for other commonly used types of servers. For example, Table 1 shows the impact of VM migration of servers from different J2EE-based tiers on the end-to-end mean response time of RUBiS [11], a widely used multitier benchmark, computed over 3 minute intervals. Futhermore, because of interference due to shared I/O, such migrations also impact the performance of other applications whose VMs run on the same physical hosts (see Section 4). Cheaper actions such as CPU tuning can sometimes be used to achieve the same goals, however. These results indicate that the careful use of adaptations is critical to ensure that the benefits of runtime reconfiguration are not overshadowed by their costs.

This paper tackles the problem of optimizing resource allocation in consolidated server environments by proposing a runtime *adaptation engine* that automatically reconfigures multitier applications running in virtualized data centers while considering adaptation costs and satisfying response-time-based SLAs even under rapidly changing workloads. The problem is challenging—the costs and benefits of reconfigurations are influenced not just by the software component targeted, but also by the reconfiguration action chosen, the application structure, its workload, the original configuration, and the application's SLAs.

To address these challenges, we present a methodology that uses automatic offline experimentation to construct cost models that quantify the degradation in application performance due to reconfiguration actions. Using previously developed queuing models for predicting the benefits of a new configuration [8], we show how the cost models allow an analysis of cost-benefit tradeoffs to direct the online selection of reconfiguration actions. Then, we develop a best-first graph search algorithm based on the models to choose optimal sequences of actions. Finally, experimental results using RUBiS under different workloads derived from real Internet traces show that our cost-sensitive approach can significantly reduce SLA violations, and provide higher utility as compared to both static and dynamic-reconfiguration-based approaches that ignore adaptation costs.

2 Architecture

We consider a consolidated server environment with a pool of physical resources H and a set of multitier applications S. We focus only on a single resource pool in this paper—a cluster of identical physical servers (hosts). Each application s is comprised of a set N_s of component tiers (e.g., web server, database), and for each tier n, a replication level is provided by reps(n). Each replica n_k executes in its own Xen VM [12] on some physical host, and is allocated a fractional share of the host's CPU, denoted by cap(n_k), that is enforced by Xen's credit-based scheduler. Therefore, system *configurations* consist of: (a) the replication degree
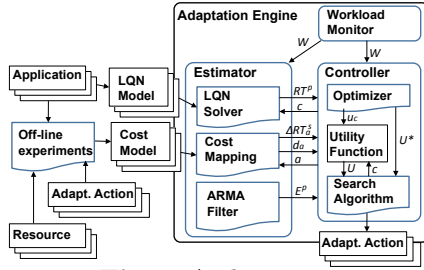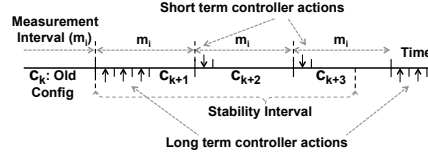
Fig. 1. Architecture   **Fig. 2.** Control Timeline

of each tier of each application, (b) the name of the physical machine that hosts each replica VM, and c) the fractional CPU capacity allocated to the replica.

Each application is also associated with a set of transaction types T_s (e.g., home, login, search, browse, buy) through which users access its services. Each transaction type t generates a unique call graph through some subset of the application tiers. For example, a search request from the user may involve the web-server making a call to the application server, which makes two calls to the database. The workload for each application is then defined as a vector of the mean request rate for each transaction type, and the workload for the entire system as the vector of workloads for each application.

We associate each application with an SLA that specifies the expected service level in the form of a target mean response time for each transaction, and the rewards and penalties for meeting or missing the target response time, as computed over a pre-specified *measurement interval*. The rewards and penalties can vary according to the application workload, thus giving rise to a step-wise *utility function* that maps mean response time and workload to a utility value that reflects the revenue gained (or lost) during the measurement interval. Using other SLA metrics does not fundamentally alter our approach.

To decide when and how to reconfigure, the adaptation engine estimates the cost and the potential benefit of each adaptation in terms of changes in the utility. Since the utility is a function of the mean end-to-end response time, the *cost of adaptation* for a given adaptation depends on its duration and impact on the applications' response times. On the other hand, the *benefit of adaptation* depends on the change in applications' response times and how long the system remains in the new configuration.

The adaptation engine manages the shared host pool by performing various *adaptation actions* such as CPU capacity tuning, VM live-migration, and component replication. As shown in Figure 1, it consists of a workload monitor, estimator, and controller. The workload monitor tracks the workload at the ingress of the system as a set of transaction request rates for each hosted application. The estimator consists of an LQN solver, a cost mapping, and an ARMA filter. The LQN solver uses layered queuing models [13] described in Section 3 to estimate the mean response time RT^s for each application given a workload W and configuration c. The cost mapping uses cost models to estimate the duration d_a and performance impact ΔRT_a^s of a given adaptation a. Both types of models are constructed using the results of an off-line model parametriza-

tion phase. Finally, the ARMA (auto-regressive moving average) filter provides a prediction of the *stability interval* E^p that denotes the duration for which the current workload will remain stable.

The controller invokes the estimator to obtain response time and cost estimates for an action's execution, which it uses to iteratively explore candidate actions. Using a search algorithm and the utility function, the controller chooses the set of actions that maximizes the overall utility. The search is guided by the upper bound on the utility U^* calculated using a previously-developed offline optimization algorithm [8] that provides the configuration that optimizes utility for a given workload without considering reconfiguration cost.

To balance the cost accrued over the duration of an adaptation with the benefits accrued between its completion and the next adaptation, the algorithm uses a parameter, called the *control window*, that indicates the time to the next adaptation. Adaptations occur only because of controller invocations. If the controller is invoked periodically, the control window is set to the fixed inter-invocation interval. If the controller is invoked on demand when the workload changes, the control window is set to the stability interval prediction E^p provided by the ARMA filter. An adaptation is only chosen if it increases utility by the end of the control window. Therefore, a short control window produces a conservative controller that will typically only choose cheap adaptation actions, while a longer control window allows the controller to choose more expensive adaptations.

Multiple controllers, each with different control windows can be used in an hierarchical fashion to produce a multi-level control scheme operating at different time-scales, and with different levels of aggressiveness. Our implementation of the adaptation engine uses a two-level hierarchical controller to achieve a balance between rapid but cheap response to short term fluctuations and more disruptive responses to long term workload changes (Figure 2). The *short term* controller is invoked periodically once every measurement interval, while the *long term* controller is executed on-demand when the workload has changed more than a specified threshold since the last long term controller invocation. To avoid multiple controller executions in parallel, the timer tracking the short term controller's execution is suspended while the long term controller is active.

3 Technical Approach

In this paper, we consider five adaptation actions: increase/decrease a VM's CPU allocation by a fixed amount, addition/removal of the VM containing an application tier's replica, and finally, migration of a replica from one host to another. Replica addition is implemented cheaply by migrating a dormant VM from a pool of VMs to the target host and activating it by allocating CPU capacity. A replica is removed simply by migrating it back to the standby pool. Some actions also require additional coordination in other tiers, e.g., changing the replication degree of the application server tier requires updating the front-end web servers with new membership.

Models. Our approach for cost estimation is based on approximate models that are constructed using off-line experimental measurements at different representative workloads using the following process. For each application s, workload w, and adaptation action a, we set up the target application along with a background application s' such that all replicas from both applications are allocated equal CPU capacity (40% in our experiments). Then, we run multiple experiments, each with a random placement of all the replica VMs from both applications across all the physical hosts. During each experiment, we subject both the target and background application to the workload w, and after a warm-up period of 1 minute, measure the end-to-end response times of the two applications $RT^s(w)$ and $RT^{s'}(w)$. Then, we execute the adaptation action a, and measure the duration of the action as $d_a^s(w)$, and the end-to-end response times of each application during adaptation as $RT_a^s(w)$ and $RT_a^{s'}(w)$. If none of application s's VMs are colocated with the VM impacted by a, no background application measurements are made. We use these measurements to calculate a delta response time for the target and the background applications, or $\Delta RT_a^s = RT_a^s - RT^s$ and $\Delta RT_a^{s'} = RT_a^{s'} - RT^{s'}$. These deltas along with the action duration are averaged across all the random configurations, and their values are encoded in a cost table indexed by the workload.

When the optimizer requires an estimate of adaptation costs at runtime, it measures the current workload w and looks up the cost table entry with the closest workload w'. To determine the impact of the adaptation a on its target application s, it measures the current response time of the application as RT^s and estimates the new response time during adaptation as $RT_a^s(w) = RT^s(w) + \Delta RT_a^s(w')$. For each application s' whose components are hosted on the same machine targeted by a, it calculates the new response times as $RT_a^{s'}(w) = RT^{s'}(w) + \Delta RT_a^{s'}(w')$. Although this technique does not capture fine-grained variations due to the difference between configurations or workloads, we show in Section 4 that the estimates are sufficiently accurate for making good decisions.

To estimate the potential benefits of a reconfiguration action, we use previously developed layered queuing network models. Given a system configuration and workload, the models compute the expected mean response time of each application. A high-level diagram of the model for a single three-tier application is shown in Figure 3. Software components (e.g., tier replicas) are modeled as FCFS queues, while hardware resources (e.g., hosts, CPU, and disk) are modeled as processor sharing (PS) queues. Interactions between tiers that result from an incoming transaction are modeled as synchronous calls in the queuing network. We account for the I/O overhead imposed by the Xen Dom-0 hypervisor, known to have significant impact (e.g., [14]), via a per-network-interaction VM monitor (VMM) delay. Although this effect impacts all VMs on the host, we model it on a per-VM basis to reduce the time to solve the model. Section 4 shows that the models provide sufficient accuracy despite this approximation.

The parameters for the models, i.e., the call graph for each transaction and the per-transaction service times at the CPU, network, disk, I/O queues at the various tiers are measured in an off-line measurement phase. In this phase, each
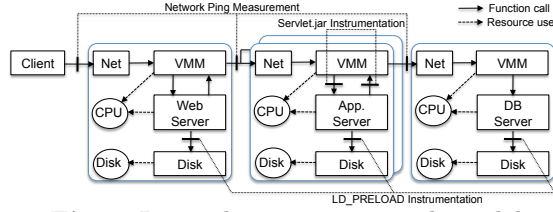
Fig. 3. Layered queueing network model

application is deployed both with and without virtualization and instrumented at different points using system call interception and JVM instrumentation. It is then subjected to test transactions, one request at a time, and measurements of the counts and delays between incoming and outgoing messages are used to parameterize the LQNS model. The models are then solved at runtime using the LQNS analytical solver [13]. More details can be found in [8].

Estimating Stability Intervals. The *stability interval* for an application s at time t is the period of time for which its workload remains within a band of $\pm b$ of the measured workload W_t^s at time t. This band $[W_t^s - b, W_t^s + b]$ is called the *workload band* B_t^s. When an application's workload exceeds the workload band, the controller must evaluate the system for potential SLA misses. When the workload falls below the band, the controller must check if other applications might benefit from the resources that could be freed up. Both cases can entail reconfiguration. Thus the duration of stability intervals impacts the choice of actions. If the workload keeps on changing rapidly, reconfiguration actions such as live-migration and replication become too expensive because their costs may not be recouped before the workload changes again. However, if the stability interval is long, even expensive adaptations are worth considering. Therefore, good predictions of the stability interval can benefit adaptation action selection.

At each measurement interval i, the estimator checks if the current workload W_i^s is within the current workload band B_j^s. If one or more application workloads are not within their band, the estimator calculates a new stability interval prediction E_{j+1}^p and updates the bands based on the current application workloads. To predict the stability intervals, we employ an autoregressive moving averages (ARMA) model of the type commonly used for time-series analysis, e.g. [15]. The filter uses a combination of the last measured stability interval E_j^m and an average of the k previously measured stability intervals to predict the next stability interval using the Equation: $E_{j+1}^p = (1-\beta) \cdot E_j^m + \beta \cdot 1/k \sum_{i=1}^{k} E_{j-i}^m$. Here, the factor β determines how much the estimator weighs the current measurement against past historical measurements. It is calculated using an adaptive filter as described in [16] to quickly respond to large changes in the stability interval while remaining robust against small variations. To calculate β, the estimator first calculates the error ε_j between the current stability interval measurement E_j^m and the prediction E_j^p using both current measurements and the previous k error values as $\varepsilon_j = (1-\gamma) \cdot |E_j^p - E_j^m| + \gamma \cdot 1/k \sum_{i=1}^{k} \varepsilon_{j-i}$. Then, $\beta = 1 - \varepsilon_j / \max_{i=0...k} \varepsilon_{j-i}$. This technique dynamically gives more weight to the current stability interval measurement by generating a low value for β when the

estimated stability interval at time i is close to the measured value. Otherwise, it increases β to emphasize past history. We use a history window k of 3, and set the parameter γ to 0.5 to give equal weight to the current and historical error estimates.

Balancing Cost and Benefit. To convert the predicted response times to utility values, the controller first calculates the instantaneous rate at which an application accrues utility either during normal operation in a configuration c, or during the execution of an adaptation action a. To do so, it uses the SLA to get the per-application workload dependent target response times TRT^s, the per-application workload dependent reward of $R^s(W_i^s)$ that is awarded every measurement interval of length M if the target response time is met, and a penalty of $P^s(W_i^s)$ imposed if the target is not met. Therefore, if the predicted response time is RT^s, the rate u^s at which utility is accrued by application s is given by:

$$u^s = \mathbf{1}[RT^s \leq TRT^s] \cdot R^s(W_i^s)/M - \mathbf{1}[RT^s > TRT^s] \cdot P^s(W_i^s)/M \quad (1)$$

In this equation, $\mathbf{1}[\ldots]$ is an *indicator function* that returns 1 if its argument is true, and 0 otherwise. During normal operation in a configuration c, the predicted response time RT_c^s is provided by the queuing models. The cost due to adaptation action a is estimated as $RT_{c,a}^s = RT_c^s + \Delta RT_a^s$. Substituting these values instead of RT^s in Equation 1 yields u_c^s and $u_{c,a}^s$, the utility accrual rate during normal execution in configuration c, and during execution of adaptation action a starting from a configuration c, respectively.

The controller then uses the control window as an estimate of how long the system will remain in a new configuration after adaptation. The control window is statically set to the controller inter-invocation time for periodic controllers and dynamically set to the stability interval for on-demand controllers. Consider the controller at the end of measurement interval i with current configuration c_i, control window CW, and evaluating an adaptation sequence A_i represented as a series of actions $a^1, a^2, \ldots a^n$. Let $d^1, d^2, \ldots, d^n$ be the length of each adaptation action, and let $c^1, c^2, \ldots, c^n$ be intermediate configurations generated by applying the actions starting from the initial configuration c_i. Let c^0 be the initial configuration c_i and c^n be the final configuration c_{i+1}. Then, the utility is:

$$U = \sum_{a^k \in A_i} (d_{a^k} \sum_{s \in S} u_{c^{k-1}, a^k}^s) + (CW - \sum_{a^k \in A_i} d_{a^k}) \cdot \sum_{s \in S} u_{c_{i+1}}^s = U_a + U_c \quad (2)$$

The first term U_a of the equation sums up the utility accrued by each application during each action in the adaptation sequence over a period equal to its action length, and second term U_c sums the utility of the resulting configuration c_{i+1} over the remainder of the control interval.

Search Algorithm. The goal of the search algorithm is to find a configuration (and the corresponding adaptation actions) for which the utility U is maximized. Configurations must satisfy the following allocation constraints: (a) for each application, only one replica from each tier can be assigned to a host, (b) the sum of CPU allocations on a host can be at most 1, and (c) the number of VMs per host is restricted to fit the available memory on the host.
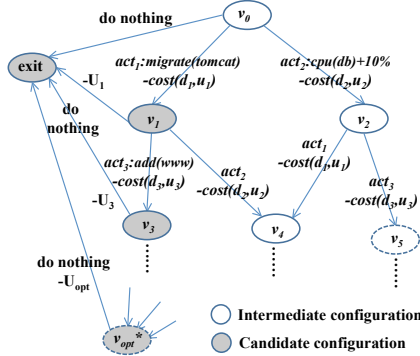
Fig. 4. Adaptation action search graph

Starting from a current configuration, a new configuration at each step is built by applying exactly one adaptation action as shown in Figure 4. The vertices represent system configurations, and the edges represent adaptation actions. We frame the problem as a shortest path problem that minimizes the negative of the utility, i.e., maximizes the actual utility. Therefore, each edge has a weight corresponding to the negative of the utility obtained while the action is being executed (i.e., $-d_a \sum_{s \in S} u^s_{c,a}$). If multiple action sequences lead to the same configuration, the vertices are combined. Configurations can be either intermediate or candidate configurations as represented by the white and gray circles in the figure, respectively. A candidate configuration satisfies the allocation constraints, while an intermediate configuration does not, e.g., it may assign more CPU capacity to VMs than is available, requiring a subsequent "Reduce CPU" action to yield a candidate configuration. Neither type of configuration is allowed to have individual replicas with CPU capacity greater than one.

Only candidate configurations have a `do nothing` action that leads the goal state, labeled as `exit` in the figure. The weight for the `do nothing` action in a configuration c is the negative of the revenue obtained by staying in c until the end of the prediction interval (i.e. $-U_c$), assuming that the best known path is used to get to c. Then, the shortest path starting from the initial configuration to the `exit` state computes the best U, and represents the adaptation actions needed to achieve optimal revenue. Intermediate configurations do not have `do nothing` actions, and thus their utility is not defined.

Although the problem reduces to a weighted shortest path problem, it is not possible to fully explore the extremely large configuration space. To tackle this challenge without sacrificing optimality, we adopt an A* best-first graph search approach as described in [17]. The approach requires a "cost-to-go" heuristic to be associated with each vertex of the graph. The cost-to-go heuristic estimates the shortest distance from the vertex to the goal state (in our case, the `exit` vertex). It then explores the vertex for which the estimated cost to get to the goal (i.e., the sum of the cost to get to the vertex and the cost-to-go) is the lowest. In order for the result to be optimal, the A* algorithm requires the heuristic to be "permissible" in that it underestimates the cost-to-go.

Input: c_i: current config., W_i: predicted workload, CW: control window length
Output: $\mathcal{A}_{opt}^i$ - the optimized adaptation action sequence
$(c^*, u^*) \leftarrow$ UtilityUpperBound (W_i)
if $c^* = c_i$ **then return** $[a_{null}]$ (do nothing)
$v^0.(a_{opt}, c, U_a, U, D) \leftarrow (\phi, c_i, 0, u^*, 0); \mathcal{V} \leftarrow \{v^0\}$
while *forever* **do**
    $v \leftarrow \text{argmax}_{v' \in \mathcal{V}} v'.U$
    if $v.a_{opt}[\text{last}] = a_{null}$ **then return** $v.a_{opt}$
    foreach $a \in \mathcal{A} \cup a_{null}$ **do**
        $v^n \leftarrow v$, $v^n.a_{opt} \leftarrow v.a_{opt} + a$
        if $a = a_{null}$ **then**
            $u_c \leftarrow$ LQNS $(W_i, v^n.c); v^n.U \leftarrow (CW - v^n.D) \cdot u_c + v^n.U_a$
        else
            $v^n.c \leftarrow$ NewConfig $(v^n.c, a); (d_a, u_a) \leftarrow$ Cost $(v^n.c, a, W_i)$
            $v^n.U_a \leftarrow v^n.U_a + d_a \cdot u_a; v^n.D \leftarrow v^n.D + d_a;$
            $v^n.U \leftarrow (CW - v^n.D) \cdot u^* + v^n.U_a$
    if $\exists v' \in \mathcal{V}$ *s.t.* $v'.c = v^n.c$ **then**
        if $v'.U > v^n.U$ **then** $v' \leftarrow v^n$
    else
        $\mathcal{V} \leftarrow \mathcal{V} \cup v^n$

Algorithm 1: Optimal adaptation search

As the cost-to-go heuristic, we use the utility u^* of the optimal configuration c^* that is produced by our previous work in [8] using bin-packing and gradient-search techniques. This utility value represents the highest rate at which utility can be generated for the given workload and hardware resources. However, it does not take into account any costs that might be involved to change to that configuration, and thus overestimates the utility that can practically be obtained in any given situation. Therefore, the utility U calculated by using u^* instead of $\sum_{s \in S} u_{c_{i+1}}^s$ in Equation 2 is guaranteed to overestimate the true reward-to-go (i.e., underestimate cost-to-go), and thus forms a permissible heuristic.

The resulting search algorithm is shown in Algorithm 1. After using the UtilityUpperBound function to compute the cost-to-go heuristic u^* for the initial configuration v^0, it begins the search. In each iteration, the open vertex with the highest value of U is explored further. New open vertices are created by applying each allowed adaptation action to the current vertex and updating $v.a_{opt}$, the optimal list of actions to get to v. When applying the do nothing action, the algorithm invokes the LQNS solver to estimate the response times of the current configuration and computes the utility. Otherwise, it invokes NewConfig to produce a new configuration and uses the cost model to compute both the adaptation cost U_a and the overall utility U as explained above. The algorithm terminates when $a.null$, i.e., "do nothing", is the action chosen.

Reducing the Search Space. The running time of the algorithm depends on the number of configurations explored by the search. The algorithm avoids lengthy sequences of expensive actions due to the optimal utility bound. However, to prevent it from getting stuck exploring long sequences of cheap actions such as CPU allocation changes, we have implemented several techniques to sig-

nificantly reduce the number of states generated without affecting the quality of the adaptations produced. The first is *depth limiting* (DL), which limits the search of paths to those of no more than n adaptation actions and effectively makes the search space finite. In our experiments, we chose $n = 7$ as the largest value that ensured that the controller always produced a decision within 30 seconds. The second is *partial order reduction* (PO), which addresses the issue that CPU tuning actions can interleave in many ways to produce the same results, but require different intermediate states, e.g., WS+10%, WS+10%, DB-10% and DB-10%, WS+10%, WS+10%. To prevent multiple interleavings without affecting the actual candidate configurations, we consider all CPU increases and decreases in a strict canonical order of components. The final technique is *action elimination* (AE), which eliminates known poor action choices, for example, disabling add replica actions when the workload for an application has diminished.

Technique	States	Time (sec)	Technique	States	Time (sec)
Naive	83497	3180	DL+PO	599	210
DL	19387	1420	DL+PO+AE	62	18

Table 2. State Space Reduction

Table 2 shows the magnitude of reductions that are achievable with these techniques using an experiment in which 10 VMs across two applications were being optimized. Adding more replicas to an application does not affect the size of the state-space. However, adding more applications does. While these results indicate that the search algorithm can be made fast enough to be used in an on-line manner while still retaining a high quality of adaptation for deployments of small to moderate size (Section 4), scalability is potentially a problem for large deployments. We are addressing this limitation in ongoing work using a combination of both better engineering and better algorithms.

4 Experimental Results

The experimental results are divided into three parts. In the first part, we describe the testbed and workloads used, and then present the measurements used in the adaptation cost models. In the second part, we evaluate the accuracy of the individual controller components: the LQNS performance models, the cost models, and the ARMA-based workload stability predictor. Finally, in the third part, we evaluate our approach holistically in terms of the quality of the adaptation decisions the controller produces and their impact on application SLAs.

4.1 Model Calibration and Testbed

Testbed. Our target system is a three-tier version of the RUBiS application [11]. The application consists of Apache web servers, Tomcat application servers, and MySQL database servers running on a Linux-2.6 guest OS using the Xen 3.2 [12] virtualization platform. The hosts are commodity Pentium-4 1.8GHz machines with 1GB of memory running on a single 100Mbps Ethernet segment. Each VM
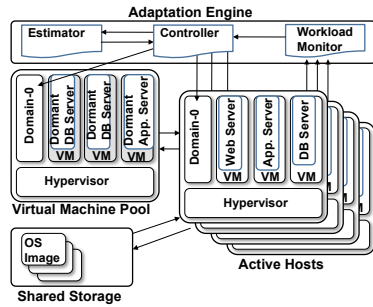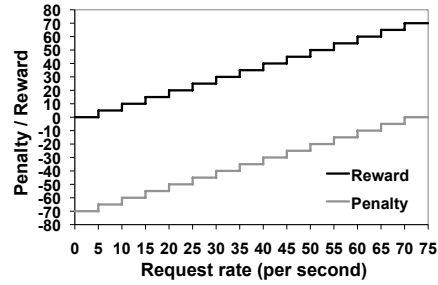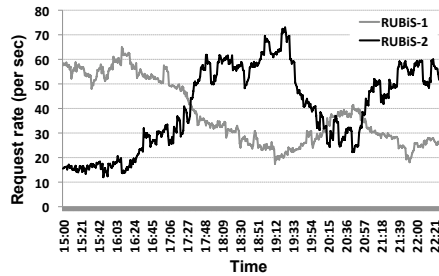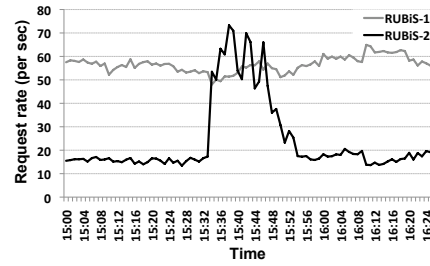
Fig. 5. Test-bed architecture

Fig. 6. SLA-based utility function

(a) Time-of-day

(b) Flash crowd

Fig. 7. Workloads

is allocated 256MB of memory, with a limit of up to 3 VMs per host. The Xen Dom-0 hypervisor is allocated the remaining 256MB. The total CPU capacity of all VMs on a host is capped to 80% to ensure enough resources for the hypervisor even under loaded conditions. Figure 5 illustrates our experimental test-bed. Four machines are used to host our test applications, while two are used as client emulators to generate workloads (not shown). One machine is dedicated to hosting dormant VMs used in server replication, and another one is used as a storage server for VM disk images. Finally, we run the adaptation engine on a separate machine with 4 Intel Xeon 3.00 GHz processors and 4 GB RAM. For MySQL replication, all tables are copied and synchronized between replicas. The Tomcat servers are configured to send queries to the MySQL replicas in a round-robin manner. We deploy two applications RUBiS-1 and RUBiS-2 in a default configuration that evenly allocates resources among all components except for the database servers, which are allocated an additional 20% CPU to avoid bottlenecks. The rewards and penalties for the applications are as specified in Figure 6 for meeting or missing a target mean response time of 84 ms in every measurement interval, respectively. The target response time was derived experimentally as the mean response time across all transactions of a single RUBiS application running in isolation in the initial configuration driven by a constant workload equal to half of the design workload range of 5 to 80 requests/sec.

Workload Scenarios. During experiments, we drive the target applications using two workloads, a time-of-day workload and a flash crowd workload. The time-of-day workload was generated based on the Web traces from the 1998
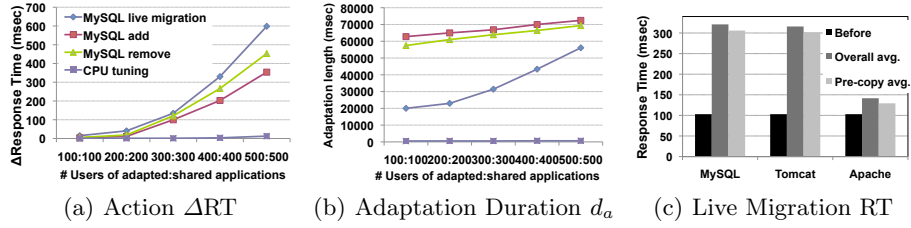
(a) Action ΔRT   (b) Adaptation Duration d_a (c) Live Migration RT

Fig. 8. Costs for various adaptation actions

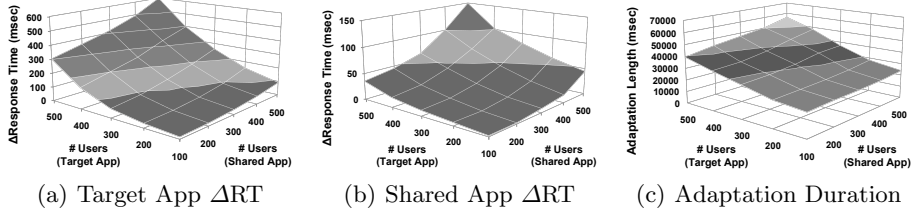(a) Target App ΔRT   (b) Shared App ΔRT   (c) Adaptation Duration

Fig. 9. Costs for MySQL live-migration

World Cup site [18] and the traffic traces of an HP customer's Internet Web server system [19]. We have chosen a typical day's traffic from each of these traces and then scaled them to the range of request rates that our experimental setup can handle. Specifically, we scaled both the World Cup requests rates of 150 to 1200 requests/sec and the HP traffic of 2 to 4.5 requests/sec to a range of 5 to 80 requests/sec. Since our workload is controlled by adjusting the number of simulated clients, we created a mapping from the desired request rates to the number of simulated RUBiS clients. Figure 7(a) shows these scaled workloads for the two RUBiS applications from 15:00 to 22:30, where RUBiS-1 uses the scaled World Cup workload profile and RUBiS-2 uses the scaled HP workload profile. The flash crowd workload shown in Figure 7(b) uses the first 90 minutes of the time-of-day workloads, but has an additional load of over 50 requests per second added to RUBiS-2 around 15:30 for a short interval.

Adaptation Costs. To measure adaptation costs, we deployed both applications and used the methodology described in Section 3. One application was the "target application" for the action, while the other was the "shared application" that was co-located with the target application, but was not reconfigured. We measured the adaptation length d_a and response time impact ΔRT_a^s for all adaptation actions and combinations of workloads ranging from 100 to 500 users for both the target and shared application. For example, Figures 8(a) and 8(b) show ΔRT_a^s and d_a for the target application when subjected to actions affecting the MySQL server and when the workload for both applications is increased equally. As is seen, ΔRT for adding and removing MySQL replicas increases as workloads increase, but the adaptation durations are not greatly affected. The costs of CPU reallocation are very small in terms of both ΔRT and d_a.

The most interesting results were those for live migration, which has been proposed in the literature as a cheap technique for VM adaptation (e.g., [10]). However, we see that live-migration can have a significant impact on a multi-tier application's end-to-end responsiveness both in magnitude and in duration.

For each of the three server types, Figure 8(c) shows the mean end-to-end response time for RUBiS measured before migration of that server, over the entire migration duration, and during the "pre-copy" phase of migration. This figure shows that although live-migration is relatively cheap for the Apache server, it is very expensive for both the Tomcat and MySQL servers. Moreover, most of this overhead incurs during the pre-copy phase. During this phase, dirty pages are iteratively copied to the target machine at a slow pace while the VM is running. In the subsequent stop-and-copy phase, the VM is stopped and the remaining few dirty pages are copied rapidly. Claims that VM migration is "cheap" often focus on the short (we measured it to be as low as 60msec) stop-and-copy phase when the VM is unavailable. However, it is the much longer pre-copy phase with times averaging 35 sec for Apache, 40 sec for MySQL, and 55 sec for the Tomcat server that contributes the most to end-to-end performance costs.

Migration also affects the response time of other VMs running on the same host. Figures 9(a) and 9(b) show the ΔRT for the target and shared applications, respectively during MySQL migration. While increases in the shared application's number of users (i.e., workload) impact the target application's response time, the target application migration has an even more significant impact on the shared application, especially at high workloads. Figure 9(c) shows how the adaptation duration increases with the target workload due to an increase in the working set memory size. In Table 3, we also show the standard deviations for these costs as percentages of the mean and calculated across all the random configurations used for measurement. The variances are quite low indicating that exact knowledge of the configuration does not significantly impact migration cost, and validating our cost model approximations. The only outlier we saw was for the response time of RUBiS-2 when two MySQL servers were co-located under high load.

Workload	Action Length	RUBiS-1 ΔRT	RUBiS-2 ΔRT
100:500	2.34%	2.95%	14.52%
300:500	7.45%	10.53%	17.14%
500:500	8.14%	6.79%	101.80%

Table 3. Variance of Adaptation Costs for MySQL Migration

4.2 Model Prediction Accuracy

We evaluate the accuracy of the LQN models and the cost models in a single experiment by using the first 220 minutes from the time-of-day workloads. Specifically, at each controller execution point and for each application, we recorded the response time predicted by the models (RT^s) for the next control interval and then compared it against the actual measured response time over the same time period. This comparison includes both the predictions of adaptation cost and performance. Figure 10 shows the results for each application. Despite the simplifications made in our cost models, the average estimation error is quite good at around 15%, with the predictions being more conservative than reality.

Second, we evaluated the accuracy of our stability interval estimation. To do this, the ARMA filter is first trained using 30 minutes of the respective
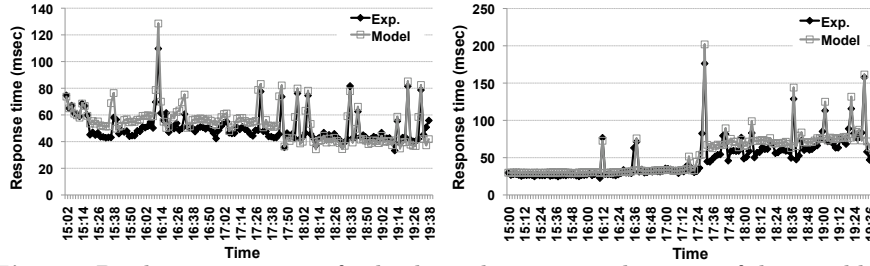
Fig. 10. Prediction accuracy for both applications under time-of-day workload

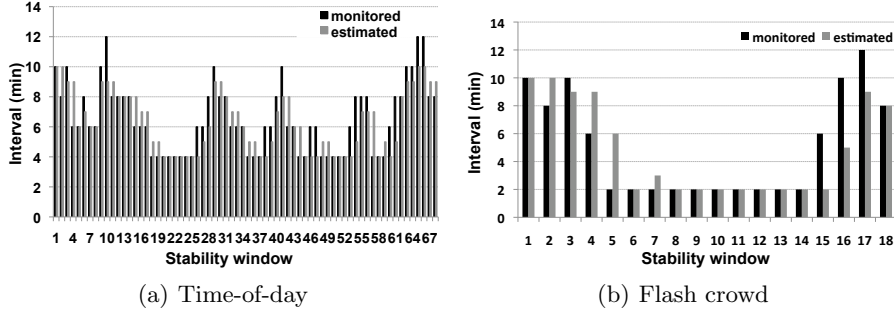
(a) Time-of-day

(b) Flash crowd

Fig. 11. Stability interval prediction error for different workloads

workloads. As shown in Figure 11(a), the filter is executed 68 times during the time-of-day experiment and provides effective estimates. The absolute prediction error against the measured interval length is around 15% for the time-of-day workloads. Meanwhile, the flash crowd workload causes an increase in the estimation error of the ARMA filter due to the short and high unexpected bursts. The results are presented in Figure 11(b). The error reaches approximately 23% because the filter over-estimates the length until the 5th stability interval when the flash crowd appears. However, the estimation quickly converges on the lower length and matches the monitored length of the stability interval until the 14th interval, when the flash crowd goes away and the filter starts to under-estimate the length. Even under such relatively high prediction errors, we show below that our cost-sensitive strategy works well.

4.3 Controller Evaluation

We evaluate our Cost-Sensitive (CS) strategy under both time-of-day workload and flash crowd scenarios by comparing its response time and utility against the following strategies: *Cost Oblivious* (CO) reconfigures the system to the optimal configuration whenever the workload changes, and uses the optimal configurations generated using our previous work [8]. *1-Hour* reconfigures the system to the optimal configuration periodically at the rate of once per hour; this strategy reflects the common policy of using large consolidation windows to minimize adaptation costs. *No Adaptation* (NA) maintains the default configuration throughout the experiment. Finally, *Oracle* provides an upper bound for utility
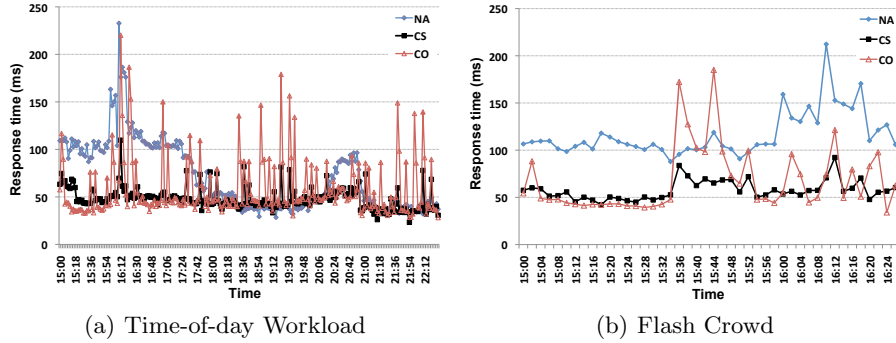
(a) Time-of-day Workload (b) Flash Crowd

Fig. 12. Response times for RUBiS-1 under different adaptation strategies

by optimizing based on perfect knowledge of future workload and by ignoring all adaptation costs.

We use the current measured workload at the controller execution point to be the predicted workload for the next control window for the CS and CO strategies. The measurement interval is set to 2 minutes to ensure quick reaction in response to workload changes. The workload monitor gets the workload for each measurement interval by parsing the Apache log file. Finally, we choose a narrow workload band b of 4 req/sec to ensure that even small workload changes will cause the controller to consider taking action.

End-to-End Response Time. First, we compare the mean end-to-end response time for all the strategies as measured at each measurement period. The results for the RUBiS-1 application are shown for the CS, CO, and NA strategies in Figures 12; the Oracle and 1-Hour plots are omitted for legibility. Figure 12(a) shows the results for the time-of-day workload. Predictably, the NA strategy is very sensitive to workload changes and shows large spikes once the workload intensity reaches the peak in both applications. For the CO and CS strategies, a series of spikes corresponds to when the adaptation engine triggers adaptations. The CS strategy has relatively short spikes and then the response time stabilizes. Meanwhile, the CO strategy has more and larger spikes than the CS strategy. This is because the CO strategy uses more adaptation actions, including relatively expensive ones such as live-migration of MySQL and Tomcat and MySQL replication, while the CS strategy uses fewer and cheaper actions, especially when the estimated stability interval is short.

Although the response time of the CO strategy is usually better than the CS strategy after each adaptation has completed, the overall average response time of CS is 47.99 ms, which is much closer to the Oracle's result of 40.91ms than the CO, 1-Hour, and NA values, which are 58.06 ms, 57.41 ms, and 71.18 ms respectively. Similarly, for the flash crowd scenario, although the ARMA filter over- and under-estimates several stability intervals, the CS strategy's mean response time of 57.68 ms compares favorably with the CO, 1-Hour, and NA values of 67.56 ms, 70.42 ms, and 116.35 ms, respectively, and is closer to the Oracle result of 40.14ms. Not surprisingly, the difference between CS and Oracle

Action	CS	CO	Action	CS	CO
CPU Increase/Decrease	14	36	Migrate (Apache replica)	4	10
Add (MySQL replica)	1	4	Migrate (Tomcat replica)	4	10
Remove (MySQL replica)	1	4	Migrate (MySQL replica)	0	2

Table 4. Total Number of Actions Triggered
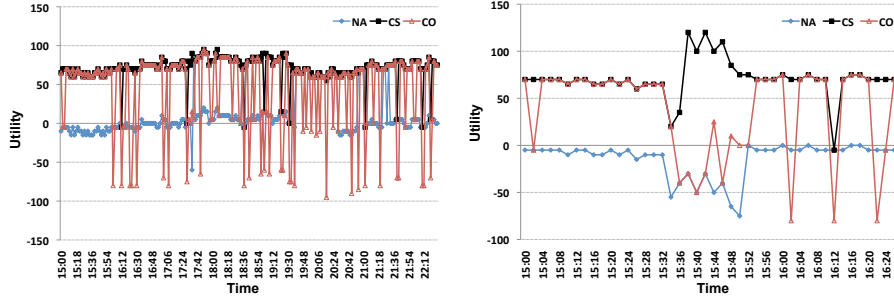
(a) Time-of-day Workload   (b) Flash Crowd

Fig. 13. Measured utility for different adaptation strategies

is larger for the flash crowd workload than the time-of-day one because the ARMA filter is wrong more often in its stability interval predictions. Also, the 1-Hour strategy does more poorly in the flash crowd case because it is unable to respond to the sudden workload spike in time. The results for RUBiS-2 were similar. Thus, the CS controller is able to outperform the CO strategy over the long run by trading-off short-term optimality for long-term gain.

To appreciate how the different strategies affect adaptation in the flash crowd scenario, consider what happens when the load to RUBiS-2 suddenly increases at 15:30. The CO controller removes a MySQL replica of RUBiS-1 and adds a MySQL replica to RUBiS-2. Meanwhile, the CS strategy also removes a MySQL replica from RUBiS-1, but then it only tunes the CPU allocation of Tomcat servers, which are much cheaper actions than adding a replica to RUBiS-2. Table 4 summarizes the number of actions of each type produced by the CS and CO strategies for the flash crowd scenario.

Utility. Using the monitored request rates and response times, we compute the utility of each strategy at every measurement interval to show the impact of adaptation actions on the overall utility. For the time-of-day workload, Figure 13(a) shows that both the CO and CS strategies have spikes when adaptation actions are triggered. However, the CO strategy has more and much deeper spikes than the CS strategy including some that lead to negative utility by violating SLAs of both applications. Meanwhile, the CS strategy chooses actions that do not violate SLAs. The utility for the flash crowd scenario in Figure 13(b) similarly shows that the CS strategy has a couple of spikes corresponding to the onset and exit of the flash crowd. However, these spikes are less severe than those of the CO strategy. The CS strategy violates the SLA of RUBiS-1 only in the measurement periods where it removes or adds a MySQL replica of RUBiS-1 (when the flash crowd starts and then after it disappears), while the CO strategy violates SLAs of both applications in many periods.

Workload	Oracle	CS	1 Hour	CO	NA
Time of day	16535	15785	10645	9280	2285
Flash Crowd	3345	3120	2035	1620	-630

Table 5. Cumulative Utility for all Strategies

We also computed the total utility accumulated over the entire experiment duration. The values for all the different strategies and workloads are shown in Table 5. Because the absolute value of the utility can differ greatly depending on the exact reward, penalty, and response time threshold values used in the SLA, it is more important to note the relative ordering between the different approaches. As can be seen, the CS strategy performs the best and has a utility very close to the Oracle for both workloads. The NA strategy predictably performs the worst. While neither the CO nor the 1-Hour strategy are competitive with CS, it is interesting to note that CO performs worse than 1-Hour. This is because CO is so aggressive in choosing optimal configurations that it incurs too much adaptation cost compared to 1-Hour, which limits adaptations to once every hour. The higher frequency of response time spikes for the CO and NA approaches indicates that this ordering is not likely to change even if a different utility function is used. These results demonstrate the value of taking workload stability and costs into account when dynamic adaptations are made.

5 Related Work

The primary contributions of this paper are (a) a model for comparing on a uniform footing dramatically different types of adaptation actions with varying cost and application performance impacts (e.g., CPU tuning vs. VM migration), and (b) considering workload stability to produce adaptations that are not necessarily optimal in the short term, but produce better results over the long run when workload variations are taken into account. We are not aware of any other work that addresses these issues, especially in the context of multitier systems with response time SLAs.

Several papers address the problem of dynamic resource provisioning [20, 21, 4, 5, 6, 7]. The authors in [7] even use queuing models to make decisions that preserve response time SLAs in multitier applications. However, none of these papers consider the performance impact of the adaptations themselves in their decision making process. The approach proposed in [22] learns the relationships between application response time, workload, and adaptation actions using reinforcement learning. It is implicitly able to learn adaptation costs as a side-benefit. However, it cannot handle never-before seen configurations or workloads, and must spend considerable time relearning its policies in case of even workload changes.

Recently, some efforts including [23, 25, 25, 27, 26] address adaptation costs. Only one adaptation action, VM migration, is considered in [23], [25], and [24]. These papers propose controllers based on online vector-packing, utilization to migration cost ratios, and genetic algorithms, respectively, to redeploy components whose resource utilization causes them to fit poorly on their current hosts

while minimizing the number or cost of migrations. Migrations are constrained by resource capacity considerations, but once completed, they are assumed not to impact the subsequent performance of the application. Therefore, the approaches cannot be easily extended to incorporate additional action types since they possess no mechanisms to compare different performance levels that could result from actions such as CPU tuning or component addition. pMapper focuses on optimizing power given fixed resource utilization targets produced by an external performance manager [27]. It relies solely on VM migration, and propose a variant of bin-packing that can minimize the migration costs while discarding migrations that have no net benefit. It also does not provide any way to compare the performance of different types of actions that achieve similar goals. Finally, [26] examines an integer linear program formulation in a grid job scheduler setting to dynamically produce adaptation actions of two types — VM migration and application reconfiguration — to which users can assign different costs. However, there is again no mechanism to compare the different performance benefits of the different actions, and the user must resort to providing a manual weight to prioritize each type of action.

In summary, the above "cost aware" approaches only minimize adaptation costs while maintaining fixed resource usage levels. They do not provide a true cost-performance trade-off that compares different levels of performance resulting from different kinds of actions. Furthermore, none of the techniques consider the limited lifetime that reconfiguration is likely to have under rapidly changing workloads and adjusts its decisions to limit adaptation costs accordingly. In that sense, they are more comparable to our "cost oblivious" policy which reconfigures the system whenever it finds a better configuration for the current workload, irrespective of future trends.

The only work we are aware of that explicitly considers future workload variations by using a limited lookahead controller (LLC) is presented in [9]. The algorithm balances application performance with energy consumption by switching physical hosts on and off. However, it only deals with a single type of coarse grain adaptation action, and requires accurate workload predictions over multiple windows into the future, something that is hard to get right. In contrast, our approach does not require any workload predictions, but can benefit from much simpler to obtain estimates of stability windows if they are available. Moreover, it is not clear whether it is practical to extend the LLC approach to allow multiple types of actions with a range of granularities.

Energy saving is considered an explicit optimization goal in [9] and [27] and is realized by shutting down machines when possible. Our current approach does not factor in the cost of energy and therefore does not consider power cycling actions. CPU power states are virtualized in [28] to produce "soft power states" exported by an hypervisor to its VMs. In this approach, each VM implements its own power management policy through the soft-states, and the management framework arbitrates requests from multiple VMs to either perform frequency scaling, or VM capacity scaling along with consolidation. It leaves policy decisions, i.e., (a) how performance goals and workload are mapped to resource

targets, and (b) when and which VMs are consolidated to which physical hosts, to the application to decide. Our goal is to automatically produce such policies.

6 Conclusions

In this paper, we have shown that runtime reconfiguration actions such as virtual machine replication and migration can impose significant performance costs in multitier applications running in virtualized data center environments. To address these costs while still retaining the benefits afforded by such reconfigurations, we developed a middleware for generating cost-sensitive adaptation actions using a combination of predictive models and graph search techniques. Through extensive experimental evaluation using real workload traces from Internet applications, we showed that by making smart decisions on when and how to act, the approach can significantly enhance the satisfaction of response time SLAs compared to approaches that do not take adaptation costs into account.

References

[1] Galletta, D., Henry, R., McCoy, S., Polak, P.: Web site delays: How tolerant are users? J. of the Assoc. for Information Sys. **5**(1) (2004) 1–28

[2] Ceaparu, I., Lazar, J., Bessiere, K., Robinson, J., Shneiderman, B.: Determining causes and severity of end-user frustration. Intl. J. of Human-Computer Interaction **17**(3) (2004) 333–356

[3] WebSiteOptimization.com: The psychology of web performance. WWW (May 2008) Accessed Apr 2009. http://www.websiteoptimization.com/speed/tweak/psychology-web-performance/.

[4] Bennani, M., Manesce, D.: Resource allocation for autonomic data centers using analytic performance models. In: Proc. IEEE ICAC, (2005) 217–228

[5] Xu, J., Zhao, M., Fortes, J., Carpenter, R., Yousif, M.: On the use of fuzzy modeling in virtualized data center management. In: Proc. IEEE ICAC, (2007) 25–34

[6] Zhang, Q., Cherkasova, L., Smirni, E.: A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In: Proc. IEEE ICAC, (2007) 27–36

[7] Urgaonkar, B., Shenoy, P., Chandra, A., Goyal, P., Wood, T.: Agile dynamic provisioning of multi-tier internet applications. ACM Trans. on Autonomous and Adaptive Sys. **3**(1) (2008) 1–39

[8] Jung, G., Joshi, K., Hiltunen, M., Schlichting, R., Pu, C.: Generating adaptation policies for multi-tier applications in consolidated server environments. In: Proc. IEEE ICAC, (2008) 23–32

[9] Kusic, D., Kephart, J., Hanson, J., Kandasamy, N., Jiang, G.: Power and performance management of virtualized computing environments via lookahead control. In: Proc. IEEE ICAC, (2008) 3–12

[10] Clark, C., Fraser, K., Hand, S., Hansen, J.G., Jul, E., Limpach, C., Pratt, I., Warfield, A.: Live migration of virtual machines. In: Proc. ACM/Usenix NSDI, (2005)

[11] Cecchet, E., Chanda, A., Elnikety, S., Marguerite, J., Zwaenepoel, W.: Performance comparison of middleware architectures for generating dynamic web content. In: Proc. ACM/IFIP/Usenix Middleware, (2003) 242–261

[12] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Wareld, A.: Xen and the art of virtualization. In: Proc. ACM SOSP, (2003) 164–177

[13] Franks, G., Majumdar, S., Neilson, J., Petriu, D., Rolia, J., Woodside, M.: Performance analysis of distributed server systems. In: Proc. Intl. Conf. on Software Quality, (1996) 15–26

[14] Govindan, S., Nath, A., Das, A., Urgaonkar, B., Sivasubramaniam, A.: Xen and co.: Communication-aware CPU scheduling for consolidated Xen-based hosting platforms. In: Proc. ACM VEE, (2007) 126–136

[15] Box, G., Jenkins, G., Reinsel, G.: Time Series Analysis: Forecasting and Control. 3 edn. Prentice Hall, (1994)

[16] Kim, M., Noble, B.: Mobile network estimation. In: Proc. ACM Conf. Mobile Computing & Networking. (2001) 298–309

[17] Russell, S.J., Norvig, P.: Artificial Intelligence: A Modern Approach. Prentice Hall (2003)

[18] Arlitt, M., Jin, T.: Workload characterization of the 1998 World Cup web site. HP Tech. Rep., HPL-99-35. (1999)

[19] Dilley, J.: Web server workload characterization. HP Tech. Rep., HPL-96-160. (1996)

[20] Appleby, K., Fakhouri, S., Fong, L., Goldszmidt, M., Krishnakumar, S., Pazel, D., Pershing, J., Rochwerger., B.: Oceano SLA based management of a computing utility. In: Proc. IFIP/IEEE IM, (2001) 855–868

[21] Chandra, A., Gong, W., Shenoy, P.: Dynamic resource allocation for shared data centers using online measurements. In: Proc. IEEE IWQoS, (2003) 155

[22] Tesauro, G., Jong, N., Das, R., Bennani, M.: A hybrid reinforcement learning approach to autonomic resource allocation. In: Proc. IEEE ICAC, (2006) 65–73

[23] Khanna, G., Beaty, K., Kar, G., Kochut, A.: Application performance management in virtualized server environments. In: Proc. IEEE/IFIP NOMS, (2006) 373–381

[24] Gmach, D., Rolia, J., Cherkasova, L., Belrose, G., Turicchi, T., Kemper, A.: An integrated approach to resource pool management: Policies, efficiency and quality metrics. In: Proc. IEEE/IFIP DSN, (2008) 326–335

[25] Wood, T., Shenoy, P., Venkataramani, A.: Black-box and gray-box strategies for virtual machine migration. In: Proc. Usenix NSDI, (2007) 229–242

[26] Garbacki, P., Naik, V.K.: Efficient resource virtualization and sharing strategies for heterogeneous grid environments. In: Proc. IFIP/IEEE IM, (2007) 40–49

[27] Verma, A., Ahuja, P., Neogi, A.: pMapper: Power and migration cost aware application placement in virtualized systems. In: Proc. ACM/IFIP/Usenix Middleware, (2008) 243–264

[28] Nathuji, R., Schwan, K.: Virtualpower: Coordinated power management in virtualized enterprise systems. In: Proc. ACM SOSP, (2007) 265–278