Generating Adaptation Policies for Multi-Tier Applications in Consolidated Server Environments

Gueyoung Jung† Kaustubh R. Joshi‡ Matti A. Hiltunen‡
Richard D. Schlichting‡ Calton Pu†

†College of Computing
Georgia Institute of Technology
Atlanta, GA, USA
{gueyoung.jung,calton}@cc.gatech.edu

‡AT&T Labs Research
180 Park Ave.
Florham Park, NJ, USA
{kaustubh,hiltunen,rick}@research.att.com

Abstract

Creating good adaptation policies is critical to building complex autonomic systems since it is such policies that define the system configuration used in any given situation. While online approaches based on control theory and rule-based expert systems are possible solutions, each has its disadvantages. Here, a hybrid approach is described that uses modeling and optimization offline to generate suitable configurations, which are then encoded as policies that are used at runtime. The approach is demonstrated on the problem of providing dynamic management in virtualized consolidated server environments that host multiple multi-tier applications. Contributions include layered queuing models for Xen-based virtual machine environments, a novel optimization technique that uses a combination of bin packing and gradient search, and experimental results that show that automatic offline policy generation is viable and can be accurate even with modest computational effort.

1 Introduction

The problem of deciding how to adapt systems to changing environments is the essence of autonomic computing. Many techniques have been proposed for such decision making: stochastic models (e.g., [9], [3], [22], [8], [20]), reinforcement learning (e.g., [19]), and control theory (e.g., [18]). Although the details are different, each follows a similar pattern: construct a parametric model of the target system (e.g., queuing model), fix some model parameters through measurement or learning, devise a strategy for optimizing the remaining parameters using the runtime state as input, implement the strategy in an *online controller* that is periodically provided with the measured runtime system state, and use the recommendations of the controller to adapt the system.

The disadvantage of online controllers is that by generating decisions algorithmically and only on demand, they may give rise to undesirable emergent properties, impede the ability of administrators to understand system behaviors, and ultimately, reduce the predictability of the target system. While some techniques—most notably control-theoretic ones—attempt to remedy concerns of unpredictable and undesired behaviors by proving stability properties of the control algorithms, by doing so they limit the system models in significant ways (e.g., through linearity assumptions) or run the risk that the guarantees are invalidated if the assumptions are not met. In contrast, rule-based expert systems address autonomic management using rules written by domain experts and executed using engines such as HP Openview. Unlike online approaches, the use of predetermined rule bases provides predictability, but with the drawback that the rules are often hard to write and cumbersome to maintain given their tight linkage to the underlying system.

We propose a novel hybrid approach for enabling autonomic behavior that provides the best of both worlds. It uses queuing models along with optimization techniques to predict system behavior and automatically generate optimal system configurations. Rather than producing these configurations on demand at runtime, they are produced offline to

This research has been partially funded by National Science Foundation grants ENG/EEC-0335622, CISE/CNS-0646430, CISE/CNS-0716484, AFOSR grant FA9550-06-1-0201, IBM SUR grants and Faculty Partnership Award, Hewlett-Packard, and Georgia Tech Foundation through the John P. Imlay, Jr. Chair endowment. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or other funding agencies and companies mentioned above.

feed a decision-tree learner that produces a compact rule set (or *adaptation policy*) that can be directly used in rule engines, audited, combined with other human-produced rules, or simply used to aid domain experts in writing and maintaining management policies. This approach of producing entire decision rule sets offline has another benefit as well—the modeling solution and optimization is entirely removed from the critical path of the system during runtime. Therefore, it is possible to model and optimize ever larger and more complex systems.

Although the approach is general, we focus on the problem of efficiently allocating resources in consolidated server environments. Server consolidation through virtualization is increasingly seen as a cost-effective way to meet the enormous demands of space, hardware, and energy of modern multi-tier enterprise systems. By hosting applications on virtual machines, resources can be shared between applications at a very fine grain (e.g., CPU cycles). However, doing so raises significant challenges such as the need to handle very different responsiveness and performance requirements of different applications, and the ability to deal with dynamic changes in resources demands as the application workloads change. Fundamentally, the management question is how to provision the applications to maximize the utility provided, while considering service level agreements (SLAs), resource availability, and workloads.

Resource provisioning in multi-tier systems is difficult even without fine grain dynamic resource allocation, and consolidated server environments make the problem even harder. The models required are more complex than in prior work on multi-tier enterprise systems (e.g., [21], [7]); not only must they accurately predict the application response time, but also its resource utilization. Furthermore, the model must factor in the impacts of the virtualization environment. Because of the multiple applications, resources, and fine-grain provisioning, the optimization space of possible system configurations is very large. Finally, because an offline policy is generated using only a subset of the possible runtime scenarios, it must be shown that any performance lost by using such an inexact policy does not invalidate the overall approach.

Overall, this paper makes two distinct contributions. First, it develops a novel approach to the construction of adaptive systems by moving the decision making process offline out of the critical runtime path, and explicitly representing automatically generated policies as human-readable rules. Second, it provides an end-to-end solution for solving the problem of dynamic resource management in virtualized, consolidated server environments hosting multiple multi-tier applications through the use of layered queuing models, a unique combination of optimization techniques, and offline rule generation.

2 Problem Statement

We begin by defining the class of applications considered, and their computing environment. Consider a pool of computing resources R and a set of multi-tier applications A. For each application $a \in A$, let N_a be the set of its constituent node types (e.g., web server, database), and for each node type $n \in N_a$, let $\mathrm{reps}(n)$ be a set of allowed replication levels. Choosing a replication level for each type results in a set N_a^k of actual nodes in the system. For example, a web application consisting of a Tomcat application server with up to 3 replicas and an unreplicated MySQL database has $\mathrm{reps}(\text{tomcat}) = \{1, 2, 3\}$ and $\mathrm{reps}(\text{mysql}) = \{1\}$. If the Tomcat server is replicated twice in a particular configuration, then the set of nodes $N_a^k = \{\text{tomcat}_1, \text{tomcat}_2, \text{mysql}_1\}$.

Each application a may also support multiple transaction types $T_a = \{t_a^1, \ldots, t_a^{|T_a|}\}$. For example, the RUBiS [4] auction site benchmark used in our testbed has transactions that correspond to login, profile, browsing, searching, buying, and selling. The workload for the application can then be characterized by the set of request rates for each of its transactions, or $w_a = \{w_a^t | t \in T_a\}$, and the workload for the entire system by $W = \{w_a | a \in A\}$.

Finally, for each application we define a utility function. Often based on SLAs, such functions can be very complex depending on the metrics they use (e.g., response time, bandwidth, throughput) and the statistics defined on the metrics (e.g., mean, 90% percentile). However, these complexities do not fundamentally alter our approach, therefore we restrict the utility to be a function of a single metric (response time) and a single statistic (mean) only. In Section 5, we outline how the approach can be extended to handle multiple metrics and statistics. Concretely, we assume that for each application a and transaction t, the SLA specifies a target response time TRT_a^t, a reward R_a^t for meeting the target, and a penalty P_a^t for missing it. Then, if RT_a^t is the actual measured response time, we define the utility for application a and transaction t as $U_a^t = w_a^t R_a^t (\mathrm{TRT}_a^t - \mathrm{RT}_a^t)$ if $\mathrm{TRT}_a^t \geq \mathrm{RT}_a^t$, and $U_a^t = w_a^t P_a^t (\mathrm{TRT}_a^t - \mathrm{RT}_a^t)$ otherwise. Other functions can also be used as long as they are monotonically non-increasing with increasing response time. Overall utility is the sum across all transactions and applications and will be denoted as U.

Given this, the goal of an adaptation policy is to configure the system such that for a given workload W, the utility U of the entire system is maximized. This maximization is carried out over the space of all possible system configurations C, where each $c \in C$ specifies: (a) the replication level $c.\mathrm{rep}(n)$ of each node n of each application a from the set $\mathrm{reps}(n)$, (b) the assignment of each replica $n_k \in N_a^k$ to a physical resource $c.r(n_k)$, and (c) the maximum cap $c.\mathrm{cap}(n_k) \in [0, 1]$ of the resource each replica is allowed

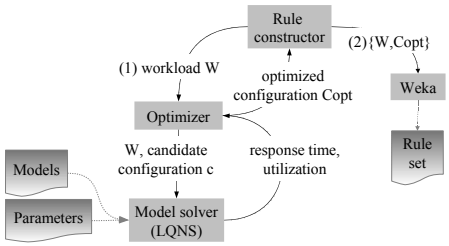


Figure 1. Approach overview

to use with the constraint that the sum of the fractional allocations across all nodes of all applications is at most 1 for each resource. Because our testbed application (RUBiS) is CPU-intensive as shown in [7], [21], and [22], the only resource type we currently consider is CPU capacity and we assume that all resources are identical. Each node replica n executes in its own Xen virtual machine [2] whose credit-based scheduling mechanism is used to enforce that the replica's CPU utilization $\rho(n_k)$ is less than the fraction $\mathrm{cap}(n_k)$ allocated to it. In Section 5, we outline how the approach can be extended to multiple resource types (e.g., CPU, disk) with varying capacities.

3 Technical Approach

Our overall approach is outlined in Figure 1. The rule set generation process is driven by the rule set constructor. It generates a set of candidate workloads and invokes the configuration optimizer to determine the best configuration c_{opt} for each candidate workload W. It then passes these workloads and associated configurations through the decision-tree learner from the Weka toolkit [1] to generate the rule sets. For each workload W passed to it, the optimizer searches through the entire configuration space for the utility maximizing configuration. To compute utility, it invokes the model solver, which for each candidate configuration c and workload W, provides an estimate of the system response time and the resource utilization of each system component. The utilization information helps the optimizer determine if the configuration is viable on the available resources or not. The model solver uses layered queuing models to predict mean response times and utilization. The response times are used to compute overall utility. The queuing models parameters are computed in an offline training phase. This section describes each component in detail, starting with the queuing models.

3.1 Application Modeling

Unlike previous work on modeling multi-tier web application that uses regular queuing networks (e.g.,[21], [20]), we chose layered queuing models [24] as the basis of our work. The reason is that in consolidated server environments with fine-grained CPU control and multiple applications, models need to be accurate over a wide range of workloads, high utilizations, and even in configurations that might be very unbalanced in terms of resource allocation amongst tiers. Thus, blocking phenomenon that are not significant in well-provisioned environments, e.g., a bottleneck due to the blocking of front-end software threads by a highly overloaded back-end server, must be explicitly modeled. Unlike standard queuing models, layered queuing networks enable such modeling by allowing multiple resources to be consumed by a request at the same time. They can be solved through a number of algorithms based on mean-value analysis (e.g., [12]). We use the LQNS modeling tool [10] as a black-box model solver.

A complication with the model is accounting for the overhead imposed by virtualization. Specifically, because Xen places device drivers for physical devices into a separate guest virtual machine called domain 0, all incoming and outgoing network communication passes through an extra "node", and incurs additional latency. Moreover, since this node potentially shares the CPU with the other virtual machines, this latency depends on both the utilization of the node and the number of messages. This additional hop is an intrinsic problem with virtualization techniques, and although system level methods to alleviate this problem have been proposed very recently (e.g., [11]), the elimination of the problem is still an open research issue. Therefore, we explicitly model and measure parameters for this virtual machine monitor delay. Note that if one does not model virtualization overhead explicitly, CPU utilization numbers and response time will be inaccurate; we are not aware of any previous modeling work that has taken this into account.

A high-level diagram of the resulting model for a single application is shown in Figure 2, and a more detailed blow-up of a portion of the model is shown in Figure 3 for 2 of the 26 transactions that comprise the RUBiS application. In the figures, the layered queuing models are specified in terms of queues or "tasks" formed by software components (depicted by parallelograms), and queues that are formed by the hardware resources (depicted by circles) that are used by the tasks. When tasks use hardware resources (depicted by dotted arrows with the label specifying the service time), or when they make synchronous calls to other tasks (depicted by solid arrows with the label specifying the average number of calls made), both the caller and the callee servers are blocked. Finally, as the detailed model shows, each task comprises of a number of "entries" (depicted as rectangles), each of which correspond to a single transaction type in the system. These entries (not their enclosing tasks) actually place demands on resources and make calls to other entries. Therefore, they are associated with parameters for service

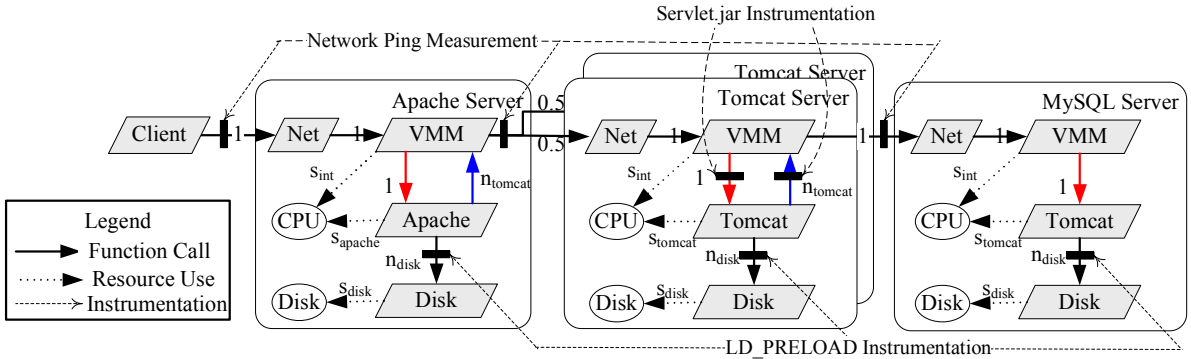


Figure 2. Layered Queuing Network model for RUBiS

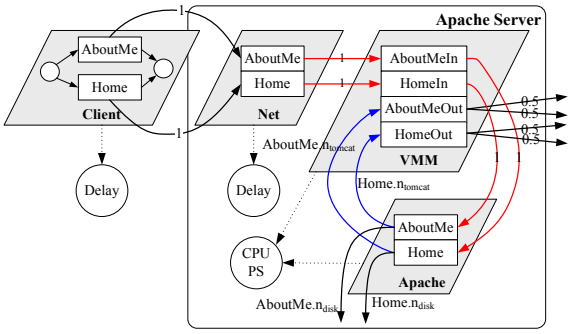


Figure 3. Detailed partial view of LQN model

time and mean number of calls made to other entries.[1]

A pre-deployment training phase facilitates collection of all the parameters required by the model. For Java servlet based applications, the parameter collection is fully automated, and does not require any instrumentation in the application code. During this phase, each application is measured in isolation with only a single replica per component, and is subjected to a workload that consists of a single transaction at a time. Multiple repetitions are done to compute mean values of the parameters. The process is then repeated for each transaction and in two environments—a virtualized environment in which each component executes in its own virtual machine, and in a native environment where each component is given its own native OS without virtualization. The solid black boxes in Figures 2 and 3 represent the points where measurements are made. A description of each task and how its parameters are computed using these measurements is as follows.

Net. Represents the latency introduced by the network. Since we assume that the network is not a bottleneck, it uses a pure delay server (i.e., no resource sharing). The service time is measured using ICMP ping measurements in the native environment.

Disk. Represents the delay due to disk I/O. To measure the service time transparently, we wrapped each component with an interception library using the `LD_PRELOAD` environment variable. The library intercepted each disk `read` and `write` call made by the application to compute the mean number of I/O calls n_{disk} and their service time.

Component (Apache, Tomcat, and MySQL). Represents the processing performed by the software component. The task is modeled using an M/M/n queue, where n is set to the number of maximum software threads each component is configured for (or ∞ in the case of MySQL, which creates threads on demand). The threads execute on a CPU queue with the processor sharing discipline (to approximate time-slice OS scheduling).

To measure the service time and number of calls for these servers transparently, we instrumented the `Servlet.jar` file that is used by every application based on Java servlets using binary rewriting. The instrumentation timestamps each incoming request from and response to the web server, and each request to and response from the database server. In addition, the client measures end to end response time for the entire transaction. Performing the experiment with only a single user at a time ensures that no queuing delay is present in the system, and the measurements at each server can be correlated. Together, their values in the native environment along with the disk I/O service times are sufficient to compute the service times for each component.

VMM. Represents the interaction delay induced by the Xen environment. We assumed the service time for this task to be equal across all machines because it is dependent on Xen and not on the application. To estimate the time, we first computed the difference between the service times of each component in the Xen environment (with the VMM task), and in the native environment (without the VMM task). Then, using knowledge of the measurement points and how many times the VMM was included in each measurement, we were able to compute the VMM service time.

Client. Generates the workload for the queuing model. Since we measure the instantaneous rates of individual transaction types at run-time, we model the workload for each application a_i as a set of $|T_i|$ independent open Pois-

[1] The figures sometimes show only a single parameter value for all the entries in a task for brevity.

son processes, one for each transaction type. This allows us to model any mix of transaction types in the workload.

The resulting models are used by the optimizer to compute utility maximizing configurations as described next.

3.2 Configuration Optimizer

The configuration optimizer performs utility maximizing replication level selection, component placement, and CPU allocation. Due to the extremely large parameter space involved and the fact that the queuing models do not have a closed form solution, the optimization task is challenging. It is easy to show that the problem is NP-Complete by a reduction to the bin-packing problem (proof is omitted for space), so an exact solution is unreasonable. However, even an approximate solution purely by traditional means, e.g., gradient search, is difficult because some of the input parameters—in particular, the choice of component placements—have very irregular effects on the utility value.

To tackle the optimization in an efficient manner, we split it into two sub-problems: selecting application configuration, which has a "regular" parameter space consisting of component replication level and CPU capacity allocation, and determining the optimal component placement for a given application configuration, which is more "irregular." For each candidate configuration generated in the configuration optimization, the component placement optimization acts as an *accept-reject* mechanism. If the "optimal" placement can fit the required components into the available resources, then the application configuration is accepted. Otherwise, it is rejected. The optimization algorithm is shown as Algorithm 1, and is explained in detail below.

Application Configuration. The application configuration optimization algorithm uses a discrete gradient-based search algorithm. Note that: (a) for any application and transaction, the utility function U is monotonically decreasing with increasing response time, (b) the response time monotonically (but not necessarily strictly) increases with a reduction in replicas of a component, (c) the response time monotonically increases with a reduction in the resource fraction allocated to the replicas of a component. Hence, if one starts off with the highest allowed replication level and a resource fraction of 1.0 for each component, the utility function would be the highest. Moreover, the algorithm can terminate as soon as an acceptable configuration is found.

Initially, the algorithm begins its search from a configuration where each node in the system is maximally replicated and assigned an entire CPU of its own irrespective of actual CPU availability. Doing so decouples the application models, and allows them to be solved independently. The LQNS solver is invoked for each application to estimate response time and the actual CPU utilization $\rho(n_k)$ of each node. The bin packer is then invoked to try to place the nodes on the available machines using the predicted CPU utilizations as the "volume" of each node. If the bin packing is unsuccessful, the algorithm re-evaluates all possible single-change degradations of the current configuration by either reducing the replication level of a single node type in some application to the next lower level, or by reducing the allowed CPU capacity for a single node in some application by a step of Δr (set to 5% by default). During reevaluation, only the model for the affected application has to be solved again, resulting in computational savings. The algorithm then picks that degraded configuration that provides the maximum reduction in overall CPU utilization for a unit reduction in utility, or gradient, which is defined as:

$$\nabla\rho = \frac{\sum_{a\in A, n_k \in N_a^k} \rho_{\text{new}}(n_k) - \rho(n_k)}{U_{\text{new}} - U} \qquad (1)$$

The whole process is repeated again until the bin packing succeeds. This technique never gets stuck because the resource fraction allocated to replicas can always be reduced to 0 to ensure that the bin packing succeeds.

Component Placement. Component placement is a performed using bin packing, which serves two purposes. First, it determines whether the total CPU consumption of all the nodes in the system fits into the available CPU capacity, and second, it also determines to which resource to assign each node. The problem of bin packing has been studied extensively in the literature, and efficient algorithms are known that can approximate the optimal solution to within

Input: W - the workload at which to optimize
Output: c_{opt} - the optimized configuration
forall $a \in A, n \in N_a$ **do**
  $c.\text{rep}(n) \leftarrow \max\{\text{reps}(n)\}, \forall n_k, c.\text{cap}(n_k) \leftarrow 1$
forall $a \in A$ **do**
  $(RT_a, \{\rho(n_k)|\forall n_k \in N_a^k\}) \leftarrow$ `LQNS` (W, a, c)
Compute U
while *forever* **do**
  $\{c.r(n_k)|\forall a, n_k\} \leftarrow$ `BinPack` $(R, \{\rho(n_k)\})$
  if *success* **then return** c
  foreach $a \in A, n \in N_a$ **do**
    $c^r \leftarrow c[\text{rep}(n) \leftarrow$ Next smallest in $\text{reps}(n)]$
    $C^k \leftarrow \{c[\text{cap}(n_k) \leftarrow$ Reduce by $\Delta r]|\forall k\}$
    foreach $c_{new} \in \{c^r\} \cup C^k$ **do**
      $(RT_a, \{\rho(n_k)\})_{new} \leftarrow$ `LQNS` (W, a, c_{new})
      Compute U_{new}, $\nabla\rho$
      if $\nabla\rho < 0 \vee \nabla\rho$ *is max so far* **then**
        $(c, \{\rho(n)\})_{opt} \leftarrow (c, \{\rho(n)\})_{new}$
      if $\nabla\rho < 0$ **then** skip to **EndRefit**
  EndRefit: $(c, \{\rho(n)\}) \leftarrow (c, \{\rho(n)\})_{opt}$

Algorithm 1: Optimal configuration generation

```
if (app1-Browse > 0.051189)
  if (app1-Browse ≤ 0.175308)
    if (app0-BrowseRegions ≤ 0.05698)
      config = "h0a0c2h1a1c2a0c0h2a0c1a1c1a1c0";
    if (app0-BrowseRegions > 0.05698)
      if (app1-Browse ≤ 0.119041)
        if (app1-Browse ≤ 0.086619)
          config = "h0a0c2h1a1c2a0c0a1c0h2a0c1a1c1";
```

Figure 4. Snapshot of a rule set

any fixed percentage of the optimal solution. In our implementation, we use the $n \log n$ time first-fit decreasing algorithm, which ensures results that are asymptotically within 22.22% of the optimal solution [13].

Using the above techniques, the optimizer is able to generate optimized configurations for a particular workload. While the generated configuration may not be provably optimal, the experimental results obtained are very good (see Section 4).

3.3 Rule-Set Construction

Finally, the highest-level component of our approach is the rule set constructor. Using the highest allowed request rate allowed for each transaction of each application (assumed to be specified in the SLA), this component first randomly generates a set of candidate workloads WS. For each workload $W \in WS$, it invokes the optimizer to find the best configuration $c_{opt}(W)$. Each configuration is encoded as a linear list of physical hosts, where each host is followed by the list of nodes that are hosted on it. Each node entry indicates the name of the application to which it belongs, followed by the name of node, and finally, the CPU capacity allocated to it. For example, the configuration `host1 app1 node1 60.0 app1 node2 40.0` indicates that the host `host1` hosts two nodes from application `app1` - `node1`, which is allocated 60% of the CPU, and `node2`, which is allocated 40% of the CPU.

These (workload, configuration) points form a partial "raw rule set" because they are rules for each of the candidate workloads. However, a complete rule set must also contain rules that apply to workloads that are not evaluated as part of the candidate set; hence, some form of interpolation is needed. To generate a final rule set, we use the J48 decision tree learner of the Weka machine learning toolkit. The generated decision tree has conditions of the form $w_a^t \leq \text{threshold}$ or $w_a^t > \text{threshold}$ at each of its branches, where w_a^t is the request rate for transaction t of application a. Each leaf of the tree encodes the configuration that should be used if all the conditions along the path that leads from the root of the tree to that leaf are satisfied.

The decision tree construction serves multiple functions.

	RT	Reward/Penalty			
Transaction Type	(s)	A_1	A_2	A_3	A_4
Browse*, Home	1	5	2	10	20
Search*	2	10	5	20	40
View*	4	10	5	20	40

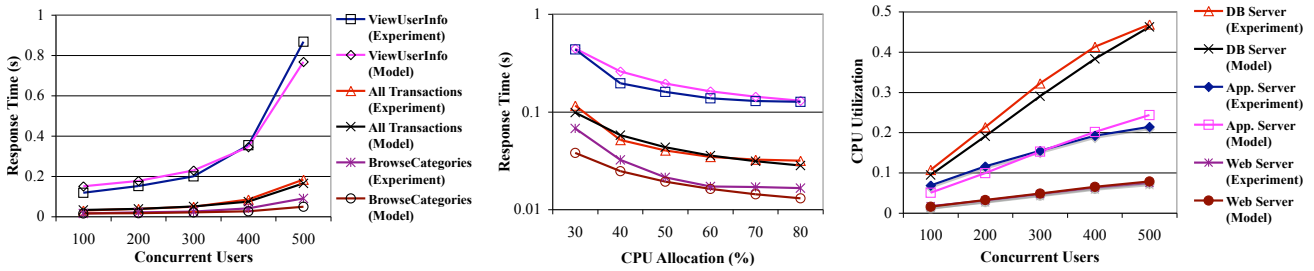
Table 1. Application SLAs

First, it provides the interpolation that is needed for rule sets to be applicable not just to the points evaluated by the optimizer, but to any workload in the range allowed by the SLAs. Second, the decision tree can be trivially linearized into a nested "if-then-else" rule set that requires less expertise to understand than the models. An example of such a rule set produced by our implementation is shown in Figure 4. Third, due to the finite number of leaves in the decision tree, all the configurations the system might include are known before deployment. This knowledge provides a degree of predictability and verifiability that is desired for business-critical systems. Finally, the tree provides compaction of the raw rule set table since the learning algorithms aggregate portions that share similar configurations and prune outliers. As a consequence of compaction and the limited number of training points, some loss of utility is expected. The next section evaluates the severity of the loss and shows that even with a modest number of training points, accurate rule sets can be constructed.

4 Evaluation

The goal of the evaluation is to demonstrate the feasibility and accuracy of the three steps in our approach: modeling, optimization, and rule set construction. Specifically, we show that (a) the constructed models accurately predict both response time and CPU utilization, (b) the optimizer produces configurations that are close to optimal, and (c) the resulting rule sets prescribe close to optimal configurations for any given workload.

Experimental Setup The target application used in our experiments is RUBiS [4], a Java-based auction system commonly used as a benchmark for multi-tier enterprise applications. To run instances of RUBiS, we used three physical hosts each with an Intel Pentium 4 1.80GHz processor, 1 GB RAM, and a 100 Mb Ethernet interface. We used the open-source version of the Xen 3.0.3 to build the virtualization environment. Linux kernel 2.6.16.29 was installed as a guest OS in each domain of Xen. Apache 2.0.54, Tomcat 5.0.28, and MySQL 3.23.58 were used as the web server, servlet container, and database server respectively in 3-tier configurations of RUBiS. Each replica was installed in its own virtual machine. The concurrency



(a) Effect of Load (b) Effect of CPU Allocation (c) CPU Utilization

Figure 5. Comparison of model vs. experimental results

parameter `maxClient` for the Apache servers was set to 335 and `maxThreads` for the Tomcat servers was set to 655 to avoid unnecessary thread induced blocking. We increased the heap size of the Tomcat server to 512 MB to avoid slowdowns induced by garbage collection and enabled `db_connection_pool`. Finally, we ran the optimization process on a machine with 4 Intel Xeon 3.00GHz processors and 4 GB RAM.

For our evaluation scenarios, we use four applications A_1, A_2, A_3, and A_4, each of which is an instance of RUBiS. RUBiS implements 26 different transaction types, but we only used the 9 transaction types included in the "Browsing" mix in our experiments. Table 1 lists the transaction types used and their SLAs for the different applications. Each application has its own SLA in terms of rewards and penalties, but all applications use the same response time thresholds.

Model Validation Our approach requires that the models be accurate enough to predict both the end-to-end response time of the system and the CPU utilizations of the different system components with different workloads and different configurations. Figure 5(a) demonstrates the accuracy of response time prediction for different transaction types and different workloads without replication and a CPU fraction of 55% for all components. The figure illustrates that the response times predicted by the model correspond well with the measured response times.

Figure 5(b) presents similar results when the CPU fractions of all components were adjusted from 30% to 80%. We set the workload (i.e., the number of concurrent users) to 200. Figure 5(c) presents the predicted CPU utilization versus the measured CPU utilization at the three tiers as the workload increases. Each component was running on its own virtual machine with a 55% CPU fraction and no replication was used. Overall, these figures demonstrate that the models are reasonably accurate and can be used as the foundation for generating adaptation rules.

Optimization Process Recall that the optimization process starts from a "maximal" configuration and reduces it by a certain CPU fraction and number of replicas at each step. In our evaluation scenario, we set the maximal configuration to one replica for the web server, two replicas for the application server, and two replicas for the database server, so that when deploying four applications, the total number of replicas is 20. For each replica, the initial CPU fraction is set to 80% which is the maximum allowed by Xen.

We used two methods to evaluate the accuracy of the optimization process. First, to evaluate global properties of the solution, we evaluated how the configuration c_o chosen by the optimizer compares with randomly generated configurations. We generated 20000 random configurations and measured their utility relative to the optimized utility of $c_o = 2274.94$. Figure 6 shows the results with the utility X on the x-axis the probability that the utility of a configuration is greater than X on the y-axis. The expanded view of the tip of the distribution shows that only very few configurations—indeed 1 out of 20000—have a utility greater than c_o. In addition, note that most configurations have poor utilities and that the distribution has several sharp steps indicating the presence of clusters of configurations with similar utilities.

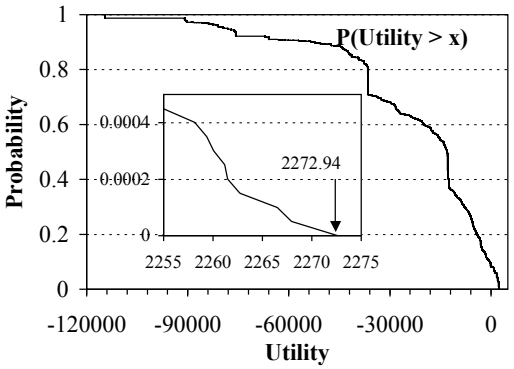


Figure 6. Global quality of optimizer

Second, we compared the response time and utility of an optimized configuration with slightly modified configurations to test for a local optimum. Specifically, we computed utility of the optimized configuration c_o for two applications with a fixed workload. We then modified the configurations slightly and used the models to predict both

the response times of the applications and the overall utility of the system. Table 2 shows the results. The notation "{APi,TCi,DBi}{↑, ↓}5%" indicates that the CPU fraction of the Apache server, Tomcat server, or MySQL server of A_i is increased or reduced by 5% (e.g., from a CPU fraction of .35 to .30). The results indicate that in all cases, any configuration change from c_o causes a response time increase in at least one application, and that in all cases except one, the overall utility is reduced compared to c_o. Furthermore, in the exceptional case, utility only increases by 0.01%.

Δ Configuration	ΔRT (%)			ΔUtility (%)
	A_1	A_2	Avg.	
1. BD2↓5%, AP2↑5%	0	1.37	0.76	-0.05
2. DB2↑5%, AP2↓5%	0	44.53	24.65	-0.29
3. TC2↓5%, TC1↑5%	-6.16	8.30	1.84	0.01
4. TC2↑5%, TC1↓5%	10.30	-4.90	1.89	-0.06
5. TC2↓5%, AP1↑5%	-1.82	8.30	3.78	-0.02
6. TC2↑5%, AP1↓5%	3.40	-4.90	-1.20	-0.01

Table 2. Local quality of optimizer

Finally, Table 3 illustrates the impact of varying CPU reduction step sizes on the cost of optimization and the utility of the optimized configuration. The former is represented by the rows labeled "Running time" and "Configurations evaluated," and the latter by "App. response time" and "Utility." The numbers in the table are the average costs for calculating the optimized configuration for two applications and one workload point (25 requests/sec for each application). The results demonstrate that the difference in accuracy is negligible across the three different step sizes even though the execution time increases significantly.

Rule set Construction The rule set that specifies the adaptation policy is constructed from randomly chosen workloads and optimized configurations. The goals of this phase are (a) to minimize loss of utility compared to the optimal, and (b) to minimize the size of the resulting rule set. These goals may conflict. For example, the size of the rule set can be reduced by merging configurations that are similar but not identical, but this adversely affects optimality.

CPU Reduction Step	1%	5%	10%
Running time (sec)	52.5	12.3	7.2
Configurations evaluated	245	53	29
App. response time (sec)	0.03327	0.03342	0.03350
- difference (%)	-	0.45	0.69
Utility	1038.45	1038.44	1038.27
- difference (%)	-	0.001	0.018

Table 3. Execution time and accuracy

Figure 7 presents the accuracy of the rule set as a function of the number of training data points used in its construction. Specifically, for a set of 100 randomly chosen workloads in each scenario, the configurations prescribed by the rule set are compared against the configurations generated by the optimizer. The differences in total utility are reported as "Utility Error." The workload points used to evaluate the accuracy are not the same as those used to construct the decision tree. As the figure shows, the error decreases as the number of data points increases, but is larger for more complex configurations. This is because with a larger number of applications, the points evaluated in the training set are a much smaller fraction of the overall workload space. However, the utility error is small in all cases: less than 1% for the 2 and 3 application configurations and approaching 4% for the 4 application configurations when the number of training set points is 3000.

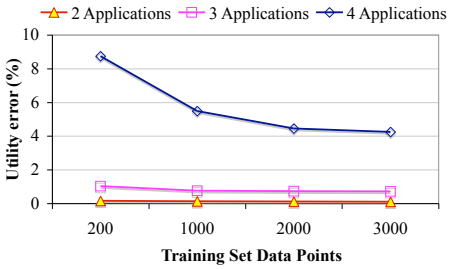


Figure 7. Accuracy of rule set

We also evaluated the size of the resulting rule set as a function of the number of applications and training data points. The results are presented in Figure 8. The dotted line denotes the line where the size of the rule set would be the same as the number of training points (x=y). The results indicate that the size of the rule set depends on the complexity of the configuration as well as the number of training points, but that the size of the rule set does not increase at the same rate as the training set.

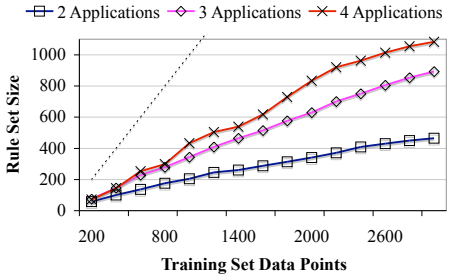


Figure 8. Size of rule set

5 Extensions

In this section, we discuss how our approach can be extended to a) allow complex utility functions incorporating multiple metrics and statistics, and to b) allow management

of multiple types heterogenous resources. Since the optimizer uses models to evaluate the utility function, the main consideration in tackling the first problem is the types metrics and statistics that can be predicted by the queuing models. Without modification, the queuing models shown in Section 3.1 predict response time, throughput, CPU utilization, disk utilization, and I/O throughput. They can also be easily extended to predict network bandwidth. However, since we do not model failures or attacks, the models cannot predict dependability oriented metrics such as availability or security. That is left for future work. If a statistic other than the mean value of a metric is required (e.g., fraction of requests for which response time is greater than some threshold t), then the models have to be solved by simulation to get accurate answers. The LQNS tool suite provides a simulator `lqsim` that can produce higher order statistics. When simulation is needed, a hybrid solution such as ours can become the only practical approach since simulation usually is usually not practical in a purely online setting.

In order to manage multiple heterogenous resource types, the bin-packing algorithm used by the optimizer must be extended to generate component placements with additional constraints. In particular, to allow for resources with different capacities, one can use one of several approximation algorithms for the variable sized bin packing problem (e.g., [14]). In order to incorporate multiple resource types (e.g., disk or network bandwidth in addition to CPU capacity), algorithms for the vector bin packing problem (e.g., [6]) can be used. Bounds for both problems have been extensively studied, and a primary benefit of our approach is that these algorithms can be used without modification in order to generate component placements. Evaluation of these algorithms remains part of our future work.

6 Related Work

While a number of recent papers address dynamic provisioning of enterprise applications, we are not aware of any that addresses the complete problem of fine-grain dynamic provisioning of such applications in virtualized consolidated server environments. For example, [19] proposes a reinforcement learning approach to resource allocation, but only for coarse provisioning at the host level (i.e., no resource sharing) and for single node applications, while [5] deals with fine-grain resource allocation, but only for single server systems that can be described using closed form performance prediction equations. Multi-tier applications are considered and queuing models are used in [21] and [22], but both consider only coarse-grain provisioning at the host level, and do not consider what to do when sufficient resources are not available. The work closest to ours is [3], which uses queuing models and a beam search to do resource allocation in data centers, but the authors do not address fine-grain resource sharing and it is not trivial to extend their optimization approach to do so. Moreover, all the above approaches are based on online control and do not consider offline policy generation.

In the area of queuing models for multi-tier systems, there is too much work to list comprehensively. Closely related, however, is [16], which uses LQN models of EJB-based enterprise systems for manual capacity planning. These models do not consider virtualized environments, which limits their applicability here since virtualization can have a performance impact in transaction-based applications [15] [23]. Approach presented in [7] deals with static provisioning of multi-tier applications, develops queuing models and uses a Xen environment, but it makes no special provisions for the virtualization environment in the model and relies on extensive experimental measurement (service times at many different CPU allocations). Black-box linear models for CPU utilization control in virtualized environments are used in [17], but these models do not make any considerations for virtualization, and are even more dependent on extensive experimentation. Although experimental techniques require less knowledge about the system, they do not scale as the number of applications and tiers increases.

Also related is work on optimization problems arising in multi-tier enterprise systems. However, few consider the problem of dynamic provisioning. For example, [27] uses optimization to determine per-transaction service times in a queuing network when they are not directly measurable, while [25] proposes efficient search algorithms and uses them to determine what experiments to conduct to choose appropriate application parameters. Our approach could also utilize such search-based methods. Finally, [8] addresses dynamic resource allocation in multi-tier virtualized service hosting platforms. It uses a capacity manager that executes periodically and reassigns resources by evaluating a model consisting of multi-tier M/M/1 queues and solves an optimization problem. However, as an online technique, it does not have the other benefits of our approach.

Machine learning and especially decision trees have been used for learning autonomic behaviors. For instance, [20] uses these to predict thresholds when a system is likely to fail its service level agreement obligations. However, most of the previous work uses decision trees in their traditional role of learning classifiers based on experimental data. We are not aware of any other work that uses decision trees to generate adaptation or management policies. A two-level control model is presented in [26], where local controllers use fuzzy control and continuous learning to determine new resource requirements given the new workload and the global controller allocates requested resources to maximizes profits.

7 Conclusions

The design of optimal, or even good, adaptation policies is one of the biggest challenges in building complex autonomic systems. This paper presents a novel hybrid approach for automatic generation of adaptation policies that uses a combination of offline model evaluation, optimization, and decision tree learning. The result is a rule set that can be inspected by human system administrators and used directly with current rule-based system management engines. We demonstrate the approach in a server consolidation scenario, where multiple multi-tier enterprise applications execute on a shared resource pool. The adaptation policies dictate replication degrees for the tiers, component placement, and virtual machine parameters. We demonstrate that it is possible to model such systems with sufficient accuracy, that our heuristic optimization technique identifies optimal or close to optimal configurations, and that the rule sets generated are accurate.

References

[1] Weka. WWW: http://www.cs.waikato.ac.nz/ml/weka.

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

[3] M. Bennani and D. Manesce. Resource allocation for autonomic data centers using analytic performance models. In *Proc. 2^{nd} Autonomic Comp. Conf.*, pages 217–228, 2005.

[4] E. Cecchet, A. Chanda, S. Elnikety, J. Marguerite, and W. Zwaenepoel. Performance comparison of middleware architectures for generating dynamic web content. In *Proc. 4^{th} Middleware Conf.*, 2003.

[5] A. Chandra, W. Gong, and P. Shenoy. Dynamic resource allocation for shared data centers using online measurements. In *Proc. IWQoS*, pages 381–400, 2003.

[6] C. Chekuri and S. Khanna. On multidimensional packing problems. *SIAM J. Comput.*, 33(4):837–851, 2004.

[7] Y. Chen, S. Iyer, X. Liu, D. Milojicic, and A. Sahai. SLA decomposition: Translating service level objectives to system level thresholds. In *Proc. ICAC.*, 2007.

[8] I. Cunha, J. Almeida, V. Almeida, and M. Santos. Self-adaptive capacity management for multi-tier virtualized environments. In *Proc. 10^{th} Symp. on Integrated Network Mgmt.*, pages 129–138, 2007.

[9] L. Franken and B. Haverkort. The performability manager. *IEEE Network*, 8(1):24–32, Jan 1994.

[10] G. Franks, S. Majumdar, J. Neilson, D. Petriu, J. Rolia, and M. Woodside. Performance analysis of distributed server systems. In *Proc. 6^{th} Conf. on Software Quality (6ICSQ)*, pages 15–26, 1996.

[11] S. Govindan, A. Nath, A. Das, B. Urgaonkar, and A. Sivasubramaniam. Xen and co.: Communication-aware CPU scheduling for consolidated Xen-based hosting platforms. In *Proc. 3^{rd} Conf. on Virtual Execution Environments*, pages 126–136, 2007.

[12] P. A. Jacobson and E. D. Lazowska. The method of surrogate delays: Simultaneous resource possession in analytic models of computer systems. *SIGMETRICS Perform. Eval. Rev.*, 10(3):165–174, 1981.

[13] E. G. C. Jr., G. Galambos, S. Martello, and D. Vigo. *Du, D.Z., Paradalos, P.M., eds.: Handbook of Combinatorial Optimization*, chapter Bin Packing Approximation Algorithms: Combinatorial Analysis. Kulwer, 1998.

[14] J. Kang and S. Park. Algorithms for the variable sized bin packing problem. *European Journal of Operational Research*, 144(2):365–372, 2003.

[15] Y. Koh, R. Knauerhase, M. Bowman, Z. Wen, and C. Pu. An analysis of performance interference effects in virtual environments. In *Proc. IEEE Symp. on Performance Analysis of Systems & Software*, pages 200–209, 2007.

[16] T. Liu, S. Kumaran, and Z. Luo. Layered queuing models for enterprise javabean applications. In *Proc. Enterprise Distributed Object Comp. Conf.*, pages 174–178, 2001.

[17] P. Padala, K. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. In *Proc. EuroSys*, pages 289–302, 2007.

[18] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus. Using control theory to achieve service level objectives in performance management. In *Proc. Symp. on Integrated Network Mgmt.*, May 2001.

[19] G. Tesauro, N. Jong, R. Das, and M. Bennani. A hybrid reinforcement learning approach to autonomic resource allocation. In *Proc. 3^{rd} Autonomic Comp. Conf.*, pages 65–73, 2006.

[20] Y. B. Udupi, A. Sahai, and S. Singhal. A classification-based approach to policy refinement. In *Proc. 10^{th} Symp. on Integrated Network Mgmt.*, pages 785–788, 2007.

[21] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi. An analytical model for multi-tier internet services and its applications. In *Proc. SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 291–302, 2005.

[22] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal. Dynamic provisioning of multi-tier internet applications. In *Proc. 2^{nd} Autonomic Comp. Conf.*, pages 217–228, 2005.

[23] Z. Wang, X. Zhu, P. Padala, and S. Singhal. Capacity and performance overhead in dynamic resource allocation to virtual containers. In *Proc. 10^{th} Symp. on Integrated Mgmt.*, pages 149–158, 2007.

[24] C. M. Woodside, E. Neron, E. D. S. Ho, and B. Mondoux. An "active server" model for the performance of parallel programs written using rendezvouz. *J. Systems and Software*, pages 125–131, 1986.

[25] B. Xi, Z. Liu, M. Raghavachari, C. Xia, and L. Zhang. A smart hill-climbing algorithm for application server configuration. In *Proc. WWW Conf.*, pages 287–296, 2004.

[26] J. Xu, M. Zhao, J. Fortes, R. Carpenter, and M. Yousif. On the use of fuzzy modeling in virtualized data center management. In *Proc. 4^{th} Autonomic Comp. Conf.*, 2007.

[27] L. Zhang, C. Xia, M. Squillante, and W. N. M. III. Workload service requirements analysis: A queueing network optimization approach. In *Proc. 10^{th} Symp. on Modeling, Analysis, and Simulation of Computer and Telecomm. Systems*, pages 23–32, 2002.