RC24136 (W0612-046) December 12, 2006
Computer Science

IBM Research Report

A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation

Gerald Tesauro, Nicholas K. Jong[1], Rajarshi Das, Mohamed N. Bennani[2]
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

[1]Department of Computer Sciences
University of Texas
Austin, TX 78712

[2]Department of Computer Science
George Mason University
Fairfax, VA 22030

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation

Gerald Tesauro*, Nicholas K. Jong†, Rajarshi Das* and Mohamed N. Bennani‡
*IBM TJ Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532 USA
Email: {gtesauro,rajarshi}@us.ibm.com
†Dept. of Computer Sciences,Univ. of Texas, Austin, TX 78712 USA
Email: nkj@cs.utexas.edu
‡Dept. of Computer Science, George Mason Univ., Fairfax, VA 22030 USA
Email: mbennani@gmu.edu

***Abstract*— Reinforcement Learning (RL) provides a promising new approach to systems performance management that differs radically from standard queuing-theoretic approaches making use of explicit system performance models. In principle, RL can automatically learn high-quality management policies without an explicit performance model or traffic model, and with little or no built-in system specific knowledge. In our original work [1], [2], [3] we showed the feasibility of using online RL to learn resource valuation estimates (in lookup table form) which can be used to make high-quality server allocation decisions in a multi-application prototype Data Center scenario. The present work shows how to combine the strengths of both RL and queuing models in a hybrid approach, in which RL trains offline on data collected while a queuing model policy controls the system. By training offline we avoid suffering potentially poor performance in live online training. We also now use RL to train nonlinear function approximators (e.g. multi-layer perceptrons) instead of lookup tables; this enables scaling to substantially larger state spaces. Our results now show that, in both open-loop and closed-loop traffic, hybrid RL training can achieve significant performance improvements over a variety of initial model-based policies. We also find that, as expected, RL can deal effectively with both transients and switching delays, which lie outside the scope of traditional steady-state queuing theory.**

I. Introduction

The primary goal of research in autonomic computing is to reduce as much as possible the degree of human involvement in the management of complex computing systems. Ideally a human would only specify a broad high-level objective as input to the system's management algorithms. Then while the system is running, the management algorithms would continually sense the system state and execute management actions that optimally achieve the high-level objective. There has been a great deal of recent research focused on algorithms that make use of explicit system performance models, such as control-theoretic or queuing-theoretic models. These approaches have achieved noteworthy success in many specific management applications. However, we note that the design and implementation of accurate performance models of complex computing systems can be highly knowledge-intensive and labor-intensive, and moreover, may require original research. For example, queuing network models of multi-tier internet services have only recently been published in [4]. Given the central goal of autonomic computing, it is therefore worth investigating whether the development of management algorithms may itself be automated to a considerable extent.

In very recent work [1], [2], [3], [5] a radically different approach based on Reinforcement Learning (RL) has been proposed for automatically learning management policies. (By "policy" we mean a mapping from system states to management actions.) In its most basic form, RL provides a knowledge-free trial-and-error methodology in which a learner tries various actions in numerous system states, and learns from the consequences of each action [6]. RL can potentially learn decision-theoretic optimal policies in dynamic environments where the effects of actions are Markovian (i.e. stationary and history-independent). In addition to firm theoretical support in the MDP (Markov Decision Process) case, there have also been many notable successful applications of RL over the last decade in real-world problems ranging from helicopter control to financial markets trading to world-championship game playing [7], [8], [9].

From an autonomic computing perspective, the RL approach offers two major advantages. First, RL does not require an explicit model of either the computing system being managed or of the external process that generates workload or traffic. Second, by its grounding in MDPs, the theory underlying RL is fundamentally a sequential decision theory that properly treats dynamical phenomena in the environment, including the possibility that a current decision may have delayed consequences in both future rewards and future observed states. This means that RL could potentially outperform other methods that treat dynamical effects only approximately, or ignore them altogether (e.g. traditional steady-state queuing theory), or cast the decision making problem as a series of unrelated instantaneous optimizations.

While RL thus offers tremendous potential benefits in autonomic computing, there are two major challenges in using it to obtain practical success in real-world applications. First, RL can suffer from poor scalability in large state spaces, particularly in its simplest and best understood form in which a lookup table is used to store a separate value for every possible state-action pair. Since the size of such a table increases exponentially with the number of state variables, it can quickly become prohibitively large in many real applications. Second,

the performance obtained during live online training may be unacceptably poor, both initially and during an infeasibly long training period. Two factors may contribute to the poor performance: (i) in the absence of domain knowledge or good heuristics, the initial RL state may correspond to an arbitrarily bad initial policy; (ii) in general RL procedures also need to include a certain amount of "exploration" of actions believed to be suboptimal. Typical mechanisms for exploration involve randomized action selection and may be exceedingly costly to implement in a live system.

In this paper, we present a new hybrid method combining the advantages of both explicit model-based methods and *tabula rasa* RL in order to address the above practical limitations. Instead of training an RL module online on the consequences of its own decisions, we propose offline training on data collected while an externally supplied initial policy (based e.g. on an appropriate queuing model) makes management decisions in the system. The theoretical basis for this approach lies in the convergence proofs of TD (Temporal Difference) learning and related methods [6], combined with Bellman's policy improvement theorem [10]. These works suggest that, given enough training samples, RL can converge to the correct value function V^{π} associated with any fixed policy π, and that the new policy whose behavior greedily maximizes V^{π} is guaranteed to improve upon the original policy π. We assume that the initial model-based policy is good enough to give an acceptable level of performance, but that there is still room for improvement. By utilizing such a policy and training offline, we avoid poor system performance that could occur using live online training. We also note that our method can be applied for multiple iterations: after we train an improved policy π' based on data obtained while π is running, we then use π' in the system to collect a second data set, which can then be used to train a further improved policy π'', etc..

The other key ingredient in our methodology is using a nonlinear function approximator in place of a lookup table to represent the value function. We have chosen to use neural networks (multi-layer perceptrons) as they have the most successful track record in RL applications, but of course many other function approximators (e.g. regression trees, CMACs, SVMs, wavelets, regression splines, etc.) could also be used. Function approximators provide a mechanism for generalizing training experience across states, so that it is no longer necessary to visit every state in the state space. Likewise they also generalize across actions, so that the need for exploratory off-policy actions is also greatly reduced. In fact we find in our system that we can obtain improved policies without any exploration, by training solely on the model-based policy decisions.

We have implemented and tested our hybrid RL approach within a realistic prototype Data Center, in which servers are to be dynamically allocated among multiple web applications so as to maximize the expected sum of SLA payments in each application. Our prototype system has been described in detail in our prior work [11], [1], [2] and we have substantial experience in developing a variety of effective policies for server allocation within this system. For the experiments reported here, we use both open-loop and closed-loop traffic scenarios. In each scenario, we first implement appropriate queuing models using standard practices for model design and parameter estimation. We then collect system performance data using a variety of initial allocation policies, including not only our best queuing model policy, but also several inferior policies (e.g. using queuing models with suboptimal parameter tunings). As a worst-case example we also use a uniform random allocation policy. For each initial policy, the collected data is used to train a corresponding neural network, which is then implemented in the prototype and tested for performance improvements. In each case we find that the RL-trained neural nets give substantially better performance compared with the corresponding initial policies. We have also obtained a number of interesting insights as to how the outperformance is obtained, particularly regarding how the RL nets are better able to deal with dynamic consequences of reallocation, such as transients and switching delays.

The rest of the paper is organized as follows. Section II describes details of our prototype Data Center. Section III describes our specific RL methodology, including an overview of the specific learning algorithm that we use (Sarsa(0)), and a summary of our prior research using tabular online RL. Section IV presents the new hybrid RL approach. Section V gives details on our initial queuing model policies regarding model design and parameter estimation. Section VI gives performance results as well as providing insight into how the trained RL value function are able to outperform the original queuing models. Conclusions and prospects for future work are given in Section VII.

II. Prototype Data Center Overview

Our prototype Data Center [11], illustrated in Figure 1, models how a set of identical servers might be dynamically allocated among multiple web applications hosted within the Center. Each application has its own Application Manager, which is responsible for performance optimization within the application and communicating with a Resource Arbiter regarding resource needs. In our model, the optimization goal within each application is expressed by a local performance-based objective function, which we call "expected business value." The Resource Arbiter's goal is to allocate servers so as to maximize the sum of expected business value over all applications (this implies that all local value functions share a common scale). Allocation decisions are made in fixed five-second time intervals as follows:

Each Application Manager i computes and reports to the Arbiter a utility curve $V_i(\cdot)$ estimating expected business value as a function of number of allocated servers. We assume that V_i expresses net expected revenue (payments minus penalties) as defined by a local performance-based Service Level Agreement (SLA). (More generally, we would also expect V_i to include other considerations such as operational cost, availability, service consistency, etc..) Upon receipt of the utility curves from each application, the Arbiter then solves

for the globally optimal allocation maximizing total expected business value (i.e. total SLA revenue) summed over the applications. (This is a polynomial time computation since the servers are homogeneous.) The Arbiter then conveys a list of assigned servers to each application, which are then used in dedicated fashion until the next allocation decision.

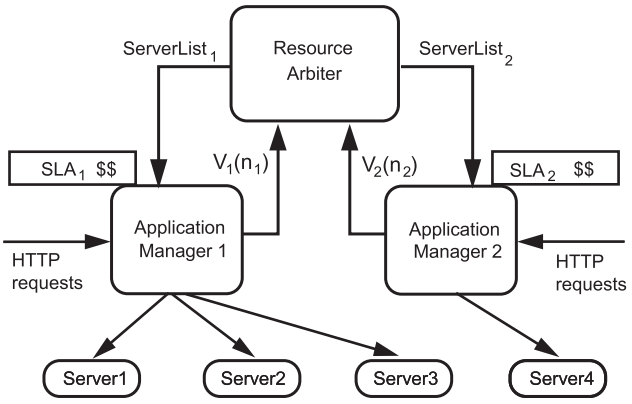

Fig. 1. Data center architecture.

Our prototype system runs on a cluster of identical IBM eServer xSeries 335 machines running Redhat Enterprise Linux Advanced Server. Our standard experimental scenario uses three applications and eight servers. Two of the applications (called $T1$ and $T2$ hereafter) are separate instantiations of "Trade3" [12], a realistic simulation of an electronic trading platform, designed to benchmark web servers. This transactional workload runs on top of IBM WebSphere and DB2. The SLA for each Trade3 application is a sigmoidal function of mean response time over the allocation interval, ranging from a maximum value of +50 to a minimum value of -150.

Demand in each Trade3 environment is driven by a separate workload generator, which can be set to operate either in open-loop or closed-loop mode. The open-loop mode generates Poisson HTTP requests with an adjustable mean arrival rate ranging from 10-400 requests/sec. In closed-loop mode, the generator simulates an adjustable finite number of customers (ranging from 5-90) behaving in closed-loop fashion, all of which have exponentially distributed think times with a fixed mean $Z = 0.17$ seconds. It is interesting to study both modes as they have very different characteristics (e.g., relationship between response time and throughput) and require rather different modeling techniques.

To provide a realistic emulation of stochastic bursty time-varying demand, we use a modified time series model of Web traffic, originally developed by Squillante et al. [13] to reset by a small increment every 1.0 seconds either the closed-loop number of customers, or the open-loop mean arrival rate. The routing policy within each Trade3 application is round-robin among its assigned servers, leading to approximately equal load balancing.

The third application in our standard scenario is a long-running, parallelizable "Batch" workload that can be paused and restarted on separate servers as they are added and removed. This emulates a non-web-based, CPU intensive computation such Monte Carlo portfolio simulations. Since there is no notion of time-varying demand in this application, we posit the Batch SLA is a simple increasing function of number of assigned servers, ranging from a value of -70 for zero servers to a maximum value of +68 for six servers[1].

III. Background on Reinforcement Learning

Reinforcement Learning (RL) refers to a set of general trial-and-error methods whereby an agent can learn to make good decisions in an environment through a sequence of interactions. The basic interaction consists of observing the environment's current state, selecting an allowable action, and then receiving an instantaneous "reward" (a scalar measure of value for performing the selected action in the given state), followed by an observed transition to a new state. An excellent general overview of RL is given in [6].

The particular RL rule we use here is an algorithm known as Sarsa(0), which learns a value function $Q_\pi(s,a)$ estimating the agent's long-range expected value starting in state s, taking initial action a and then using policy π to choose subsequent actions [6]. (For simplicity we hereafter omit the π subscript.) Sarsa(0) has the following form:

$$\Delta Q(s_t,a_t) = \alpha(t)[r_t + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)] \qquad (1)$$

Here (s_t,a_t) are the initial state and action at time t, r_t is the immediate reward at time t, (s_{t+1},a_{t+1}) denotes the next state and next action at time $(t+1)$, the constant γ is a "discount parameter" between 0 and 1 expressing the present value of expected future reward, and $\alpha(t)$ is a "learning rate" parameter, which decays to zero asymptotically to ensure convergence.

When $Q(s,a)$ is represented using a lookup table, equation 1 is guaranteed to converge for MDP environments, provided that the policy for action selection is either stationary, or asymptotically "greedy," i.e. it chooses the action with highest Q-value in a given state. However, as detailed below, these conditions do not strictly hold in our system in three respects: (i) Our applications are not exactly Markovian, although this may be a reasonable approximation. (ii) In our hybrid approach, described in Section IV, we use function approximation instead of lookup tables. (iii) Our formulation of RL is not at the global decision maker level, but instead localized within each application, and from the local perspective the global policy need not be greedy or stationary. We chose this formulation due to much better scalability to many applications, as well as a basic design principle that the arbiter should not receive detailed state descriptions from each application. The issue of whether localized RL converges in such a "composite MDP" scenario [14] is an interesting open research topic which is discussed in more detail in [2].

A. Summary of Previous RL Approach

In our previous work [1], [2], [3] we implemented a localized version of online RL within the Trade3 application

[1]We enforce a constraint that each Trade3 must have at least one server, so that Batch can never be allocated more than six servers.

manager. The RL module observed the application's local state, the local number of servers allocated by the arbiter, and the reward specified by the local SLA. A lookup table was used to represent $Q(s,a)$. Due in part to the table's poor scalability, we made a severe approximation in representing the application state solely by the (discretized) current mean arrival rate λ of page requests, and ignoring several other sensor readings (e.g. mean response times, queue lengths, number of customers, etc.) that could also have been used. Hence our value function was two-dimensional: $Q = Q(\lambda, n)$, and it was encouraging that RL could achieve comparable performance to standard queuing models using such a simple function.

Since our learning was online and influenced the arbiter's decision making through the reported value estimates, it was important that $Q(\lambda, n)$ be initialized to values that would yield an acceptable initial performance level of the arbiter's policy at the start of the run. For this purpose we chose a heuristic initialization assuming a linear dependence of Q on demand per server λ/n. Such initialization required a modicum of elementary domain knowledge, but perhaps would be more difficult with additional state variables or a more complex application.

We also devised two methods for dealing with significant sparsity of table cell visits observed during the learning run. First, we used a so-called "ε-greedy" exploration rule, in which the arbiter would choose a random allocation with probability $\varepsilon = 0.1$ instead of the utility-maximizing allocation. This turned out to incur minimal cost in the simple system described in [1], [2], [3] but can be expected to become more costly as the complexity of the allocation task increases. Second, we imposed soft monotonicity constraints based on the assumption that the table values should be monotone decreasing in λ and monotone increasing in n. This requires further domain knowledge to devise and implement. We presume and in fact found in our work described below that the use of function approximation can greatly reduce or eliminate the need for such techniques.

IV. Hybrid RL Approach

In our hybrid RL approach, a nonlinear function approximator is trained in batch mode on a dataset recorded while an externally supplied policy makes management decisions within a given computing system. While we have chosen to use neural networks here, due to their prior successes in RL applications as well as their robust generalization in high-dimensional spaces [15], our methodology may be generally used with other types of function approximators.

The use of an external policy is motivated by a desire to avoid poor performance that would be expected during online learning. This necessitates either using a good external policy, or initializing the RL value function in a way that implements a good initial policy. However, we expect the latter option to be quite difficult, as it most likely requires extensive domain knowledge of the particular system, and may also require deep knowledge of the function approximator methodology. Moreover, a carefully chosen RL initialization may lead to an inferior final result compared to, for example, random initialization.

The use of batch training is motivated by two factors. First, due to the sample complexity of RL, a large number of observed samples may be required before RL is capable of learning an effective policy. Second, RL is a "bootstrapping" procedure with non-stationary targets, since the target regression value for the observation at time t depends on the function approximator's estimated value for the observation at time $t+1$. Hence, as regression moves a function approximator toward a set of targets, this causes the targets themselves to change. This suggests a batch training methodology comprising a large number of sweeps through the dataset, with incremental learning in each sweep.

Our hybrid RL approach for learning a value function for an application takes as input a recorded sequence of $(T+1)$ observations $\{(s_t, a_t, r_t), 0 \le t \le T\}$ produced by an arbitrary management policy, where (s_t, a_t, r_t) are the observed state, action and immediate reward at time t. We use Algorithm 1 to compute a neural network value function based on the recorded observations.

Algorithm 1 Compute Q

```
1:  Initialize Q to a random neural network
2:  repeat
3:      SSE ← 0 {sum squared error}
4:      for all t such that 0 ≤ t < T do
5:          target ← r_t + γQ(s_{t+1}, a_{t+1})
6:          error ← target − Q(s_t, a_t)
7:          SSE ← SSE + error · error
8:          Train Q(s_t, a_t) towards target
9:      end for
10: until CONVERGED(SSE)
```

This algorithm borrows proven methods from supervised learning techniques, which repeatedly trains on a fixed set of input-output pairs until attaining some convergence criterion. For example, we observe faster learning when we randomize the order of presentation in the loop on line 4. The training procedure on line 8 uses the standard back-propagation algorithm, which adjusts each weight in the neural network in proportion to its error gradient. However, this algorithm is not an instance of supervised learning, due to the non-stationarity of targets described above. Although this property removes any theoretical convergence guarantees, we find that the mean squared error roughly decreases monotonically to a local minimum. Note that due to the stochastic gradient nature of Sarsa/back-propagation there is noise in the error measure *SSE* after each epoch. Hence the convergence criterion in line 10 must maintain sufficient history of prior *SSE* values to detect when the error reaches some asymptote.

Apart from the choice of function approximator, the most important design issue is variable selection and representation of the state-action pairs (s_t, a_t) as inputs to the function approximator. In principle the state s_t should be fully observable

in the MDP sense, i.e. it should contain all current or prior sensor readings, for both the traffic arrival process and the system service process, relevant to optimal decision making. This could be problematic in cases where there are great many potentially relevant sensor readings, and the hybrid RL user does not have sufficient systems expertise to discern which ones are most relevant. However, given our reported prior success representing system state solely by current demand λ, we suggest this is reasonable to try at least as a baseline experiment, and we use this choice once again for the more complex experiments reported in Section VI.

The above discussion suggests a two-input representation $(s,a) = (\lambda, n)$ as described previously in Section III-A. However, in the present work we are particularly interested in the dynamic consequences of allocation decisions. For example, there may be switching delays, in which a newly allocated server is initially unavailable for a certain period of time. There may also initially be transient suboptimal performance due to load rebalancing, or starting new processes or threads on the newly allocated servers. To handle such effects, we employ a "delay-aware" representation in which the previous allocation decision n_{t-1} is added to the state representation at time t. As long as such delays or transients last no more than one allocation interval, this should suffice to learn the impact of such effects on expected value, while for longer delays one would also need further historical information n_{t-2}, etc..

Using the three inputs $(\lambda_t, n_{t-1}, n_t)$, we then train a standard multi-layer perceptron containing a single hidden layer with 12 sigmoidal hidden units, and a single linear output unit. We set a back-propagation learning rate of 0.005 and typically run Algorithm 1 for $\sim$10-20K sweeps through the dataset. We set the Sarsa discount parameter $\gamma = 0.5$.

V. Initial Queuing Model Policies

To bootstrap our hybrid RL approach with reasonable initial policies we adopt some of the recently proposed model-based approaches for online performance management and resource allocation [16], [17], [18], [19]. Typically, these approaches employ steady-state queuing theory models in a dynamic environment where the model parameters are continuously updated based on measurements of system behavior. Of course, in addition to providing the initial policy, such models also provide suitable performance benchmarks for our hybrid RL approach.

To model the arrival and departure of requests in our two different workload generator modes for the Trade3 application, we construct two different types of queuing networks: open network and closed network. An open queuing network has external arrival and departure of requests from an infinite population of customers, while a closed queuing network has a finite population of customers, each alternating between the think state and the submitted state.

The underlying principles of modeling the two types of queuing network were guided by certain salient features of our prototype Data Center. The probability distribution of service times for Trade3 HTTP requests was empirically found under moderate load-conditions to be well approximated by an exponential distribution with a mean of 8.3×10^{-3} seconds. About 0.2-0.3% of requests resulted in significantly longer response times and we attribute such outliers to JVM garbage collection process.

Following [16], [17], [18], [19], we reestimate the parameters of our models at the end of each allocation interval t, based on measurements of various state variables such as the mean arrival rate λ_t, the mean response time R_t, the total number of servers allocated n_t, and the number of customers M_t. Due to the previously mentioned small change per time step generated by our time series model of workload intensity, a reasonably accurate forecast is that intensity at time $t+1$ approximately equals current intensity[2] at time t. Armed with this forecast, the models can then be used to estimate the utility curve $V_{t+1}(n_{t+1})$ for all possible values of n_{t+1}.

A. Open Queuing Network Model

Due to the round-robin assignment of the HTTP requests among the available servers, we can model an application in the open queuing network with an overall demand λ and n servers as a system of n independent and identical parallel open networks each with one server and a demand level of λ/n. We leverage this observation as well as the Poisson arrival process and the exponential service times in our experimental setup to apply the parallel M/M/1 queuing formulation to model the mean response time characteristics of an application. In the M/M/1 model, the mean response time R of a service center with a service rate μ and an arrival rate λ is given by $R = \frac{1}{\mu - \lambda}$ [20]. Therefore, given a resource level n_{t+1} for the next allocation period, a predicted mean arrival rate $\lambda_{t+1} = \lambda_t$, and assuming that the workload is uniformly divided among the n_{t+1} servers,

$$R_{t+1} = \frac{1}{\mu - \frac{\lambda_t}{n_{t+1}}}. \tag{2}$$

The unknown model parameter μ can be estimated by applying the same formula to the current allocation period, and after rearranging terms, we obtain $\mu = \frac{1}{R_t} + \frac{\lambda_t}{n_t}$. From empirical evidence, we find that the above derivation of μ is quite sensitive to variations in R_t that are caused by the finite sampling and garbage collection processes in Java. To dampen the effect of these variations, we use an exponential smoothing for μ before solving for R_{t+1} for all possible values of n_{t+1}. Experiments to validate our M/M/1 model with exponential smoothing for estimates of μ (with smoothing parameters in the range 0.1-0.5) show that under steady workload conditions the percentage difference in the means of the actual and predicted response times is less than 20% for a wide range of arrival rates and server allocations.

B. Closed-Loop Queuing Network Model

Since an application in the closed network also distributes HTTP requests in a round-robin fashion, we can model an

[2]It is possible that some slight improvement in forecast accuracy could be obtained using standard time series analysis and forecasting techniques applied to historical workload intensity variations.

application with an overall number of customers M and n servers as a system of n independent parallel closed networks each with one server and M/n customers. We leverage this observation in employing the well known Mean Value Analysis (MVA) formulation [20] to model the mean response time characteristics of an application. In order to obtain the mean response time for a given level of server allocation there are two unknown model parameters in the MVA method: the average think time Z and the average service time at the server, $1/\mu$. Using the Interactive Response Time Law, and assuming $M_{t+1} = M_t$, we can compute $Z = \frac{M_t}{X_t} - R_t$, where X_t is the measured mean overall throughput of the application. We obtain an estimate of the mean service time $1/\mu$ by using the M/M/1 model as follows: $\mu = \frac{1}{R_t} + \lambda_t$. We apply exponential smoothing (smoothing parameter in the range 0.1-0.5) on estimates of Z and μ to account for finite sampling effects and Java garbage collection. We find that under steady workload conditions, the means of the actual and predicted response times differ by less than 25% for a wide range of customer population sizes and server allocations.

VI. Results

A. Performance Results without Switching Delay

We first present results for open-loop and closed-loop systems for scenarios with no switching delays, i.e., newly assigned servers are immediately available for processing workload. The performance measure is total SLA revenue per allocation decision summed over all three applications. For a variety of initial model-based policies, we compare the initial policy performance with that of its corresponding hybrid RL trained policy.

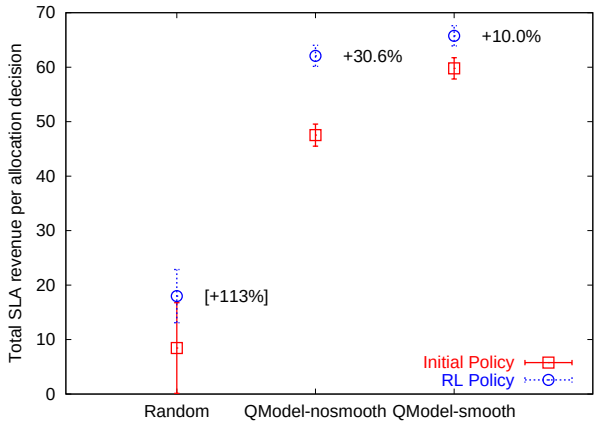

Fig. 2. Performance of various strategies in open-loop zero-delay scenario.

The results from the open-loop and closed-loop systems are shown respectively in Figures 2 and 3. The percentage figures denote relative improvement of hybrid RL policies over their corresponding initial queuing models. (For the random initial policies, such percentages are shown in brackets as they have dubious meaning in our opinion.) The error bars denote 95% confidence intervals for the reported values; this calculation does not reflect the nearly identical demand traces used in each experiment. To address this factor we also performed paired T-test when comparing the hybrid RL results with each corresponding initial policy.

In the open-loop case we examine three initial policies: our open-loop model with exponential smoothing of the estimates of μ (with a smoothing parameter of 0.1), the open-loop model without smoothing, and for a baseline comparison, a uniform random allocation policy. We see substantial improvement of each hybrid RL trained policy over its corresponding initial policy in both relative and absolute terms. For each of the three pairs of experiments, we can reject the null hypothesis that there is no difference between the means of the two performance measures using paired T-test at 1% significance level with P-value $\leq 10^{-6}$.

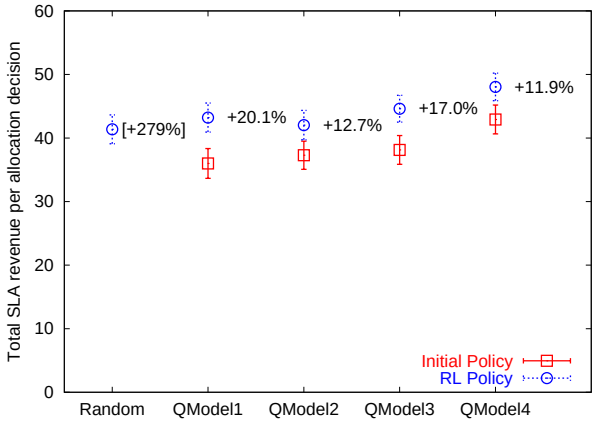

Fig. 3. Performance of various strategies in closed-loop zero-delay scenario. (The random policy performance lies off the scale at -23.0.)

In the closed-loop case we again examine a random initial policy, whose average utility score of -23.0 lies off the scale, plus four different queuing model polices. The best model, QModel4, is our MVA approach with exponential smoothing used to estimate customer think time, governed by a smoothing parameter of 0.1. QModel1 is the MVA approach with smoothing turned off. QModel2 uses the same model as QModel4 but predicts cumulative future utility instead of immediate utility. Since the model's predicted utility is steady in all future time steps, we accomplish this by rescaling its utility estimates by $1/(1-\gamma)$ with $\gamma = 0.5$ identical to the discount factor used by RL. By examining such a model we address the issue of whether RL obtains an advantage over the queuing models merely by estimating future reward instead of immediate reward. QModel3 uses the parallel M/M/1 model designed for the open-loop scenario, which ought to be wholly inappropriate here, but nonetheless provides an interesting test of training hybrid RL with a suboptimal initial model.

Once again we find substantial improvement of hybrid RL over each initial policy. In particular, the improvement over the random policy is enormous, while the improvement over the queuing models is consistently at a double-digit percentage level with high statistical significance (rejected each null hypothesis using paired T-test at 1% significance level with

P-value $\leq 4 \times 10^{-3}$).

Our general observations regarding the experiments in Figures 2 and 3 are as follows. First, our hybrid RL approach to policy improvement clearly works quite well in this domain. Second, our results are generally in accordance with prior studies of policy iteration, where one typically finds large improvement starting from weak initial policies, and progressively smaller improvement starting from stronger initial policies. Third, we have obtained several insights as to how the queuing model estimates lead to suboptimal allocations, and how the RL trained neural nets are able to do better:

One important factor is that the RL nets learn to directly estimate expected utility (i.e. SLA revenue), whereas the queuing models do so indirectly, by first estimating response time, and then estimating revenue using the SLA payment function. We find that both open-loop and closed-loop queuing models tend to overestimate the impact of server additions or subtractions on current response time. Since in most cases the application's current state corresponds to low response time and high utility, the estimation error will be considerably worse for removing servers than for adding servers, due to much flatter slope of the SLA function in the latter case. As a result the Trade3 applications tend to be slightly overprovisioned on average. However, the RL nets, by learning to estimate utility directly, are able to achieve less biased estimation errors. This leads to the Trade3 applications receiving slightly fewer servers on average, with a slight loss of Trade3 revenue, but the loss is more than made up by substantially greater Batch revenue. In terms of application performance metrics, hybrid RL policies typically obtain much better Batch throughput with little degradation in Trade3 response time. For example, after training on the best open-loop queuing model in Figure 2, hybrid RL obtains a 12.7% improvement in Batch throughput while mean Trade3 response time only increases by 2.6%.

Another important factor is that our steady-state queuing models are unable to take dynamical effects into account (although more sophisticated models could do so). However, the RL nets are able to take into account dynamic effects such as transients and switching delays, and possibly even implicit predictability of future demand in a current state, by learning policies that exhibit hysteresis. We analyze this phenomenon immediately below in Section VI-B.

B. Policy Hysteresis

We illustrate how our RL approach deals with dynamic effects by examining a value function trained on data from a system with no switching delay using our best open-loop queuing model. Figure 4 illustrates a portion of the learned joint value function, estimating total value summed over all three applications as a function of the states and allocations for each application. The plot shows how estimated total value varies as a function of demand in one Trade3 application ($T1$) while demand in the other Trade3 application is held fixed at 400 requests per second. Each curve segment corresponds to a stable joint allocation, as indicated, for which the estimated value would decrease changing from the indicated allocation

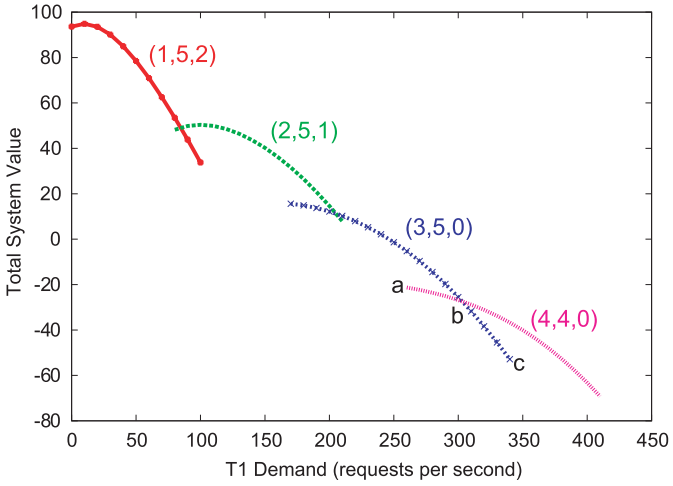

Fig. 4. Stable allocation policies $(T1, T2, Batch)$ in an RL value function.

to any other allocation. The horizontal range of each segment denotes the range of $T1$ demand values over which the allocation is stable, while the *y*-axis values denote the expected value for this allocation, given that the allocation is unchanged from the previous decision. The history dependence results from the delay-aware representation in the Trade3 neural nets, in which the expected value depends on both the previous and new allocation decision.

In the absence of history dependence, clearly there could be only one stable allocation at any given level of $T1$ demand. However, with history dependence there can be multiple stable allocations at a given demand level, and the actual allocation choice would then depend on the previous allocation.

For example, suppose the current joint allocation is $(3,5,0)$ and the current $T1$ demand is below 300 requests per second, i.e., to the left of point **b** in the figure. If $T1$ demand then increases slightly past point **b**, the allocation does not immediately switch to $(4,4,0)$ but instead remains at $(3,5,0)$. This is because the higher value shown for $(4,4,0)$ would only apply if the allocation is already $(4,4,0)$. When considering a switch from $(3,5,0)$ to $(4,4,0)$ there would be another estimated value curve (not shown), lying below the stable $(3,5,0)$ curve, incorporating an estimated cost of switching the allocation. The $(3,5,0)$ allocation would continue to be selected until $T1$ demand increases past point **c**, where the estimated value to switch to $(4,4,0)$ exceeds the value of remaining at $(3,5,0)$. Likewise if current allocation is $(4,4,0)$ and $T1$ demand drops below point **b**, the preferred allocation would be to remain at $(4,4,0)$ until point **a** is reached, where the predicted value of switching to $(3,5,0)$ would exceed the value of remaining at $(4,4,0)$.

The retardation of allocation switches with respect to instantaneous system state is an example of "hysteresis" in the allocation policy, borrowing a term describing physical systems with lagged responses. We can identify at least four mechanisms in our system which would make such hysteresis beneficial. First, there is the obvious case of switching delays, when there is a clear and direct cost of reassigning servers,

as they are unable to perform any useful work during the delay interval. Second, there may also be a transient period of suboptimal performance within an application after servers are added or removed. We observe that when a newly assigned server begins processing Trade3 requests, the performance is initially sluggish, presumably due to starting processes and creating Java threads in WebSphere. There is also a finite period of time need to rebalance the load among the new set of servers. Third, the increased need for resource motivating a potential switch may be temporary (e.g. due to a short-term demand fluctuation), leading after switching to a future cost of immediately switching back. Fourth, there is the phenomenon of thrashing, in which removing a server from an application causes it to increase its reported need for servers, so that the server is immediately switched back.

We find evidence in our prototype system that all four of the above phenomena can occur using steady-state queuing models, and that the simple delay-aware input representation used by the RL nets enables them to learn hysteretic policies that effectively deal with these phenomena. Evidence pertaining to switching delays and thrashing is presented below in Section VI-C.

C. Performance Results with Switching Delay

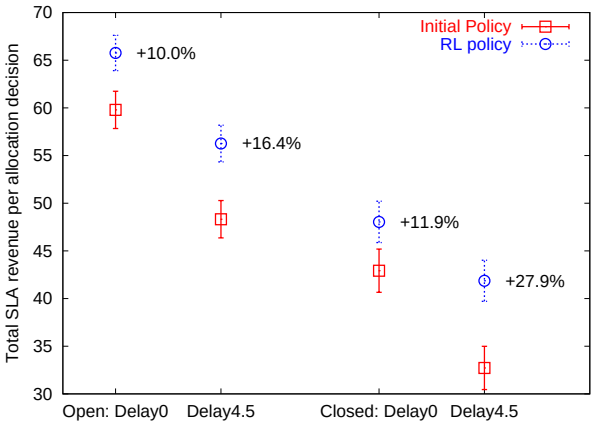

Fig. 5. Comparison of delay=4.5 sec with delay=0 results in open-loop and closed-loop scenarios.

Figure 5 presents a comparison of our zero-delay results, using our best open-loop and closed-loop queuing models, with corresponding experiments that incorporate a switching delay of 4.5 seconds upon reassigning a server to a different application. The delay is asymmetric in that the server is immediately unavailable to the old application, but does not become available to the new application until 4.5 seconds have elapsed. We chose the delay to be a huge fraction of the five second allocation interval so that empirical effects due to the delay would be as clear as possible. We see in Figure 5 that imposing this delay does in fact substantially harm the average performance in all cases. However, the amount of policy improvement of hybrid RL over its initial policy increases in both absolute and relative terms. In the open-loop scenario the improvement increases from 10.0% to 16.4%, while in the closed-loop scenario the improvement jumps from 11.9% to 27.9%.

Experiment	$< n_T >$	$< \delta n_T >$
Open-loop Delay=0 QM	2.27	0.578
Open-loop Delay=0 RL	2.04	0.464
Open-loop Delay=4.5 QM	2.31	0.581
Open-loop Delay=4.5 RL	1.86	0.269
Closed-loop Delay=0 QM	2.38	0.654
Closed-loop Delay=0 RL	2.24	0.486
Closed-loop Delay=4.5 QM	2.36	0.736
Closed-loop Delay=4.5 RL	1.95	0.331

TABLE I

MEASUREMENTS OF MEAN NUMBER OF SERVERS $< n_T >$ ASSIGNED TO A TRADE3 APPLICATION, AND MEAN CHANGE IN NUMBER OF ASSIGNED SERVERS $< \delta n_T >$ PER TIME STEP, IN THE EIGHT EXPERIMENTS PLOTTED IN FIGURE 5.

The enhanced policy improvement seen above provides one line of evidence that the RL policies effectively deal with switching delays. Other evidence of this can be seen in Table I, which exhibits basic statistics averaged over the two Trade3 applications $T1$ and $T2$ from the eight experiments shown in Figure 5. The quantity $< n_T >= (< n_{T1} > + < n_{T2} >)/2$ is the average number of assigned servers, while $< \delta n_T >= (< \delta n_{T1} > + < \delta n_{T2} >)/2$ is the RMS change in number of assigned servers from one time step to the next. As mentioned previously, the mean number of servers assigned to a Trade3 application is slightly less for the RL nets than for the queuing models, and there is a further slight reduction for the RL nets for 4.5 second delay compared to zero delay. More importantly, the $< \delta n_T >$ statistics reveal noticeably less server swapping when using RL nets compared to queuing models, with the effect becoming quite pronounced ($>\sim$50% reduction) in the 4.5 second delay case. We attribute the reduction in $< \delta n_T >$ in the latter case partly to greater stickiness or hysteresis in the RL trained value functions, and partly due to reduction or elimination of thrashing in overloaded situations. In fact, massive thrashing under very high load appears to be the main factor behind the poor performance of the closed-loop queuing model with 4.5 second delay. In this run, we found that when one of the Trade3 applications ($T1$, say) estimates that it needs seven servers to obtain high utility, and $T2$'s estimates fluctuate between needing one and two servers, the arbiter's allocation decision for ($T1$,$T2$,*Batch*) will thrash between (7,1,0) and (1,2,5) leading to huge loss of utility given the 4.5 second switching delay. However, the RL nets trained on this data set prefer to keep a steady allocation of 5 servers under a heavy demand spike, thereby eliminating completely this particular thrashing mode.

VII. CONCLUSIONS

One contribution of this paper is to devise and demonstrate success of a new hybrid learning method for resource valuation estimates, combining disparate strengths of both reinforcement learning and model-based policies, within a dynamic server allocation scenario applicable to Data Centers. Our hybrid

RL approach neatly takes advantage of RL's ability to learn in a knowledge-free manner, requiring neither an explicit system model nor an explicit traffic model, and requiring little or no domain knowledge built into either its state space representation or its value function representation. Moreover, through the use of a simple "delay-aware representation" including the previous allocation decision, our approach also naturally handles transients and switching delays, which are dynamic consequences of reallocation lying outside the scope of traditional steady-state queuing models. On the other hand, our hybrid approach also exploits the ability of a model-based policy to immediately achieve a high (or at least decent) level of performance as soon as it is implemented within a system. By running such a policy to obtain training data for RL, we maintain acceptable performance in the live system at all times, and avoid potentially poor performance that would be expected using online RL. We may also exploit robustness of model-based policies under various types of system changes, e.g. hardware upgrades or changes in the SLA, which require retraining of the RL value functions. When such changes occur, we can fall back on the model-based policy to deliver an acceptable performance level which accumulating a second training set to be used for RL retraining.

We would also like to stress, however, that it would be a mistake to view our work solely as a method for server allocation in Data Centers. Due to the broad generality of RL itself, we view hybrid RL as having potentially wide applicability throughout many different areas of systems management. The types of management applications holding the most promise for hybrid RL would have the characteristics of: (a) a tractable state-space representation; (b) frequent online decision making depending upon time-varying system state; (c) frequent observation of numerical rewards in an immediate or moderately delayed relation to management actions; (d) pre-existing policies that obtain acceptable (albeit imperfect) performance levels. Clearly there are a great many performance management applications having such properties. Among them are dynamic allocation of other types of resources, e.g., bandwidth, memory, CPU slices, threads, LPARs, etc.. We would also include performance-based online tuning of system control parameters, such as web server parameters, OS parameters, database parameters, etc.. Finally, we note that hybrid RL could conceivably go beyond performance management to encompass simultaneous management to multiple criteria (e.g. performance and availability), as long as the rewards pertaining to each criterion are on an equivalent numerical scale.

In future work we plan further investigations of the scalability of hybrid RL/function approximation as the application state space increases in size and complexity. Specifically in the Data Center scenario, we plan to add several other state variables (e.g. mean response time, mean queue lengths, etc.) to the RL input representation in order to investigate the effect on training time and sample complexity, as well as whether further performance improvements can be obtained. We will also study whether progressively better performance results can be obtained via multiple iterations of the policy improvement method. It also will probably be necessary at some point to tackle the issue of adding exploratory actions to the initial policy for general usage of hybrid RL, even though it was not necessary in our experiments. Finally we are investigating with our IBM colleagues whether there may be feasible commercial deployments of hybrid RL, for example, in WebSphere XD [21] and Tivoli Intelligent Orchestrator [22].

ACKNOWLEDGMENT

The authors would like to thank Jeff Kephart for many helpful discussions.

REFERENCES

[1] R. Das, G. Tesauro, and W. E. Walsh, "Model-based and model-free approaches to autonomic resource allcation," IBM Research, Tech. Rep. RC23802, 2005.

[2] G. Tesauro, "Online resource allocation using decompositional reinforcement learning," in *Proc. of AAAI-05*, 2005.

[3] G. Tesauro, R. Das, W. E. Walsh, and J. O. Kephart, "Utility-function-driven resource allocation in autonomic systems," in *Proc. of ICAC-05*, 2005.

[4] B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi, "An analytical model for multi-tier internet services and its applications," in *Proc. of SIGMETRICS-05*, 2005.

[5] D. Vengerov and N. Iakovlev, "A reinforcement learning framework for dynamic resource allocation: First results," in *Proc. of ICAC-05*, 2005.

[6] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press, 1998.

[7] G. Tesauro, "Temporal difference learning and TD-Gammon," *Commun. ACM*, vol. 38, no. 3, pp. 58–68, 1995.

[8] J. Moody and M. Saffell, "Learning to trade via direct reinforcement," *IEEE Transactions on Neural Networks*, vol. 12, no. 4, pp. 875–889, 2001.

[9] A. Y. Ng *et al.*, "Inverted autonomous helicopter flight via reinforcement learning," in *Intl. Symposium on Experimental Robotics*, 2004.

[10] R. E. Bellman, *Dynamic Programming*. Princeton University Press, 1957.

[11] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das, "Utility functions in autonomic systems," in *Proc. of ICAC-04*, 2004, pp. 70–77.

[12] IBM, "Websphere benchmark sample," http://www-306.ibm.com/software/webservers/appserv/benchmark3.html, 2004.

[13] M. S. Squillante, D. D. Yao, and L. Zhang, "Internet traffic: Periodicity, tail behavior and performance implications," in *System Performance Evaluation: Methodologies and Applications*, E. Gelenbe, Ed. CRC Press, 1999.

[14] S. Singh and D. Cohn, "How to dynamically merge Markov Decision Processes," in *Advances in Neural Information Processing Systems*, M. I. Jordan, M. J. Kearns, and S. A. Solla, Eds., vol. 10. MIT Press, 1998.

[15] A. R. Barron, "Complexity regularization with application to artificial neural networks," in *Nonparametric Functional Estimation and Related Topics*, G. Roussas, Ed., 1991.

[16] P. Pradhan, R. Tewari, S. Sahu, C. Chandra, and P. Shenoy, "An observation-based approach towards self-managing web servers," in *Proc. of Intl. Workshop on Quality of Service*, 2002.

[17] A. Chandra, W. Gong, and P. Shenoy, "Dynamic resource allocation for shared data centers using online measurements," in *Proc. of ACM/IEEE Intl. Workshop on Quality of Service (IWQoS)*, 2003, pp. 381–400.

[18] M. N. Bennani and D. A. Menascé, "Assessing the robustness of self-managing computer systems under variable workloads," in *Proc. of ICAC-04*, 2004.

[19] ——, "Resource allocation for autonomic data centers using analytic performance models," in *Proc. of ICAC-05*, 2005.

[20] D. A. Menascé, V. A. F. Almedia, and L. W. Dowdy, *Performance by design: Computer Capacity Planning by Example*. Upper Saddle River, NJ: Prentice Hall, 2004.

[21] IBM, "WebSphere Extended Deployment," www.ibm.com/software/webservers/appserv/extend/, 2006.

[22] TIO, "Tivoli Intelligent Orchestrator product overview," http://www.ibm.com/software/tivoli/products/intell-orch, 2005.