Online Performance Management Using Hybrid Reinforcement Learning

Gerald Tesauro and Rajarshi Das
IBM TJ Watson Research Center
Hawthorne, NY 10532

Nicholas K. Jong
Dept. of Computer Sciences
Univ. of Texas
Austin, TX 78712

Abstract

We present a new hybrid approach to performance management, combining disparate strengths of Reinforcment Learning (RL) with model-based (e.g. queuing-theoretic) approaches. Our method trains nonlinear function approximators using offline RL on data collected while a model-based policy controls the system. By training offline we avoid potentially poor performance in live online training, while function approximation allows generalization across both states and actions, so that the need for exploratory actions may be greatly reduced. Our results show that, in a prototype resource allocation scenario among multiple web applications, hybrid RL training can achieve significant performance improvements over a variety of initial queuing 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.

Introduction

Developing effective behavioral policies for real-time performance management in complex distributed computing systems is an important goal of current systems research. The predominant approach to this makes 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 development of accurate models of complex computing systems is both difficult and highly knowledge-intensive, and moreover, such modeling should become progressively more difficult as systems become increasingly complex and distributed.

This paper studies a radically different approach using Reinforcement Learning (RL), recently proposed in (Das, Tesauro, & Walsh 2005; Tesauro 2005; Tesauro *et al.* 2005; Vengerov & Iakovlev 2005). RL is natural and holds great promise for systems management applications in our view, for two reasons: (i) it can learn effective policies (e.g., optimal policies for Markov Decision Problems) in the absence of explicit system models; (ii) it uses an inherently sequential decision making approach, taking into account future consequences of a current decision.

While the studies cited above show promise, we note two potentially significant practical problems in usage of online RL. First, the above studies suggest that tens of thousands of observations may be required, implying online training times as long as several months, which is unacceptable in many applications. Second and more importantly, the performance during live online training may be unacceptably poor, due to two factors: (a) in the absence of domain knowledge or good heuristics, the initial RL state may correspond to an arbitrarily bad initial policy; (b) RL procedures generally need to include "exploration" of actions believed to be suboptimal. Typically this involves randomized action selection which may be exceedingly costly in a live system.

To address the above limitations, we devise in this paper a hybrid method combining the advantages of both explicit model-based methods and *tabula rasa* RL. Instead of online training on its own decisions, we propose offline RL training on data collected while an externally supplied initial policy (based e.g. on a good queuing model) makes management decisions in the system. Assuming the initial policy's performance level is acceptable, we are thereby assured of acceptable performance while gathering training data (apart from any additional exploratory actions).

Once the training set is acquired, offline sweeps through the dataset may be several orders of magnitude faster than the underlying physical time scales. This permits multiple sweeps through the dataset, allowing training of sophisticated nonlinear value function approximators that learn too slowly to be trained online. Function approximators provide a mechanism for generalizing training experience across states, so that it is unnecessary to visit every state in the state space. They likewise generalize across actions, so that the need for exploratory off-policy actions may be 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.

The following section describes our experimental platform for testing hybrid RL in a dynamic server allocation task involving multiple web-based workloads. Subsequent sections present details regarding the hybrid RL approach; experimental results in the testbed; and concluding remarks. Our queuing models are briefly described in an appendix.

Experimental Setup

Our experimental testbed, illustrated in Figure 1, comprises real servers and realistic Web-based workloads, and follows the scenaro in (Das, Tesauro, & Walsh 2005) for dynamically allocating a set of identical servers among multiple web applications. Each application has its own Application Manager module which communicates with a Resource Ar-

biter module regarding resource needs. Allocation decisions are made in fixed five-second time intervals as follows:

Each Application Manager i reports to the Arbiter a commonly scaled utility curve $V_i(\cdot)$ estimating expected business value as a function of number of allocated servers. We assume that an application's business value is defined by a Service Level Agreement (SLA) stipulating payments or penalties as function of one or more performance metrics. Upon receipt of the utitity curves, the Arbiter then solves for the globally optimal allocation maximizing total expected value. The Arbiter then conveys a list of assigned servers to each application, which are then used in dedicated fashion until the next allocation decision.

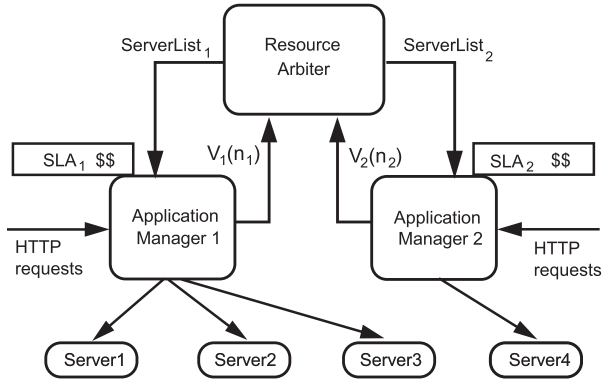

Figure 1: Resource allocation scenario.

Our standard testbed uses eight HTTP servers (3.06GHz Xeon machines) and three applications. Two of the applications are separate instantiations of "Trade3," a realistic simulation of an online trading platform. Each Trade3 SLA is a sigmoidal function of mean response time over the allocation interval. The third application is a long-running "Batch" workload that can be paused and restarted as servers are added and removed. This emulates a CPU-intensive task such as Monte Carlo portfolio simulations. Since there is no notion of time-varying Batch demand, we posit its SLA to be a simple increasing function of number of assigned servers[1].

Demand in each Trade3 environment is driven by a separate workload generator, which can operate either in open-loop or closed-loop mode. The open-loop mode generates Poisson HTTP requests with an adjustable mean arrival rate. The closed-loop mode simulates an adjustable finite number of customers behaving in closed-loop fashion, all of which have exponentially distributed think times with a fixed mean.

To emulate stochastic bursty time-varying demand, we use a modified version of the Squillante et al.(1999) Web traffic model to reset by a small increment every 1.0 seconds either the closed-loop number of customers, or the open-loop mean arrival rate. Routing within each Trade3 application is round-robin among its assigned servers.

Hybrid RL Approach

Reinforcement Learning (RL) refers to a set of methods for learning decision policies through interactions with an environment, consisting of: observing the current state, selecting an allowable action, and then receiving an instantaneous "reward" (a scalar measure of value), followed by an observed transition to a new state. In our hybrid RL procedure, sketched in Algorithm 1, training data is obtained by running an initial policy in the live system and recording a set of $(T+1)$ observations $\{(s_t,a_t,r_t), 0 \le t \le T\}$, where (s_t,a_t,r_t) are the observed state, action and immediate reward at time t. Using these data, we train a value function $Q(s,a)$ estimating long-range expected value starting in state s and taking initial action a. This value function defines a new RL-based policy which then replaces the original policy.

Algorithm 1 Hybrid RL procedure

```
1:  Initialize Q-function approximator (e.g. randomly)
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)
```

Algorithm 1 envisions batch training wherein for each observation (s_t,a_t,r_t) we compute in Line 5 a target Q-value, and then regress the current $Q(s_t,a_t)$ values toward their targets. Line 5 derives from the well-known Sarsa learning rule: $\Delta Q(s_t,a_t) = \alpha[r_t + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)]$, where α is a small learning rate constant and γ is "discount factor" between 0 and 1 expressing the present value of expected future rewards. The details of batch training will depend on the specific function approximator used. In some cases one can do incremental training per individual observation, whereas in other cases the entire batch of observations and targets may be needed, e.g., to construct a regression tree. Typically the batch training will proceed for some number of sweeps through the training set until some error based criterion (e.g., *SSE*) has reached an asymptotic value.

Experimental Implementation

In our testbed scenario, we could implement hybrid RL at the global Arbiter level, but this would scale poorly, as the global state space scales exponentially with the number of applications. Hence we adopt the decompositional approach of (Tesauro 2005) in which RL was implemented separately within each Trade3 application, using only local state and local number of allocated servers. This achieves scalability to many applications, and worked well empirically despite the lack of rigorous convergence guarantees.

For each Trade3 application, the action a_t comprises the local number of servers n_t allocated at time t. To represent the state s_t, many sensor readings could be used, but for simplicity we follow (Tesauro 2005) in using only the current demand λ_t. Additionally, in the present work we are particularly interested in the dynamic consequences of allocation decisions. For example, when a server is added there

[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.

may be initial transient suboptimal performance, or switching delays, in which the server is initially unavailable for some time. To handle such effects, we employ a "delay-aware" representation in which the previous allocation n_{t-1} is added to the state representation at time t. As long as such effects last no more than one allocation interval, this should suffice to learn the impact on expected value (longer delays would require n_{t-2}, etc.).

We represent the Q-function $Q(\lambda_t, n_{t-1}, n_t)$ with neural networks, due to their prior successes in RL applications as well as their robust generalization in high-dimensional spaces. For each Trade3 application we train a standard multi-layer perceptron containing three input units, a single hidden layer with 12 sigmoidal hidden units, and a single linear output unit. We scale the inputs to the interval $[0,1]$, and in Line 8 of Algorithm 1 we use back propagation with a learning rate of 0.005. Empirically, 10-20 thousand sweeps through the dataset suffice to achieve convergence. We set the Sarsa discount parameter $\gamma = 0.5$.

Results

Performance Results without Switching Delay

We first present results for open-loop and closed-loop systems without switching delays. 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.

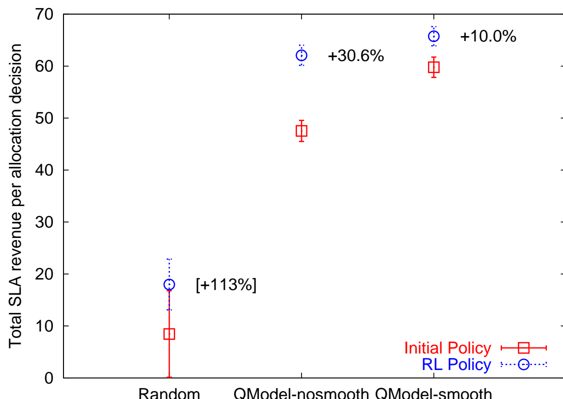

Figure 2: Performance of various strategies in open-loop zero-delay scenario.

The open-loop and closed-loop results 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 exponentially smoothed parameter estimates, 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. In each pair of experiments, a paired T-test rejects the null hypothesis that there is no difference between the two performance means at 1% significance level with P-value $\leq 10^{-6}$.

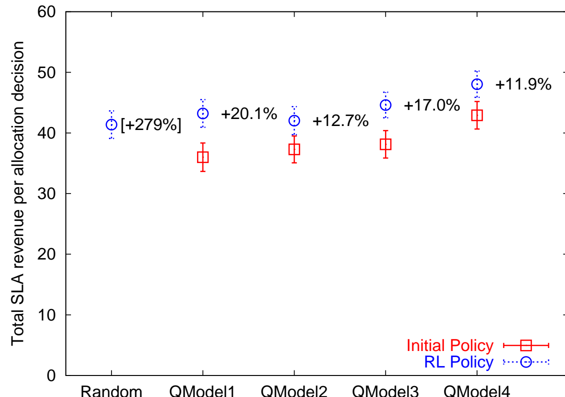

Figure 3: Performance of various strategies in closed-loop zero-delay scenario. (The random policy performance lies off the scale at -23.0.)

In the closed-loop case we again examine a random initial policy, plus four different queuing model polices. The best model, QModel4, is our Mean Value Analysis (MVA) model with exponentially smoothed parameter estimates. QModel1 is the MVA model without smoothing. QModel2 uses the same model as QModel4 but predicts cumulative future utility instead of immediate utility, i.e., its estimates are rescaled by $1/(1-\gamma)$ with $\gamma = 0.5$ identical to the RL discount factor. Such a model addresses 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 $\leq 4 \times 10^{-3}$).

Performance Results with Switching Delay

Figure 4 compares our zero-delay results, using our best open-loop and closed-loop queuing models, with corresponding experiments that impose a delay of 4.5 seconds when a server is reassigned 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 its empirical effects would be as clear as possible. We see in Figure 4 that imposing this de-

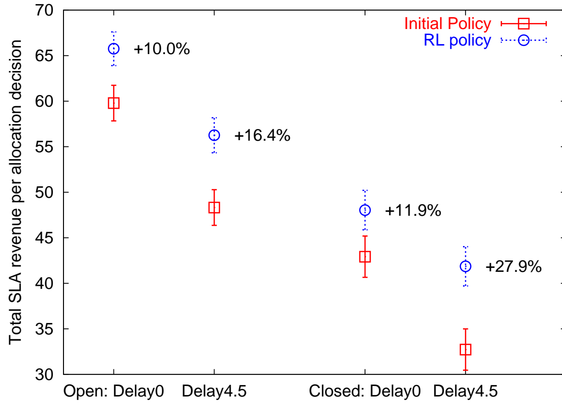

Figure 4: Comparison of delay=4.5 sec with delay=0 results in open-loop and closed-loop scenarios.

lay 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%.

Insights Into Hybrid RL Outperformance

We offer three insights as to how hybrid RL is able to outperform the initial queuing model policies. The first has to do with estimation bias. For reasons too technical to detail here, the queuing model policies end up having a bias toward overprovisioning, due to interactions of their response time estimates with the nonlinear SLA function. 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 utility, but the loss is more than made up by substantially greater Batch utility. In terms of application performance metrics, typically there is a large (double-digit percent) improvement in Batch throughput, at the cost of only a slight (few percent) worsening of Trade3 response time. For example, in the open-loop zero delay experiment with our best queuing model, the Batch throughput improves by 12.7% while the average Trade3 response time only increases by 2.6%.

The second point is that whereas steady-state queuing models are unable to properly treat transients and switching delays, our RL nets are able to do so. The learned policies exhibit "hysteresis," i.e., a tendency to prefer steady allocations over switching based on instantaneous state. Some evidence[2] for this is seen in Table 1, which exhibits basic statistics averaged over the two Trade3 applications $T1$ and $T2$ from the eight experiments shown in Figure 4. 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. We see that 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 sec 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 sec delay case.

[2] Full evidence cannot be presented due to space limitations.

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 1: 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 4.

The reduction in $<\delta n_T>$ generally reflects hysteresis in the RL policies, and specifically relates to our third insight, which is that the RL policies exhibit greatly reduced thrashing. In experiments with 4.5 sec delay, massive thrashing under high load conditions is a significant problem using the queuing model policies. An example of this from a closed-loop experiment is given in Figure 5, which shows a five-minute interval in which $T1$ is moderately loaded and $T2$ is very heavily loaded in most of the interval. The upper plot shows the queuing model allocations, while the lower plot shows the hybrid RL allocations under the same demand trace. We see much steadier allocations in the latter case. This is due partly to the RL value functions' general preference to have no more than five servers, and partly to their projected high switching cost which inhibits a large instantaneous increase in servers (say, from 2 to 5-6) within an application.

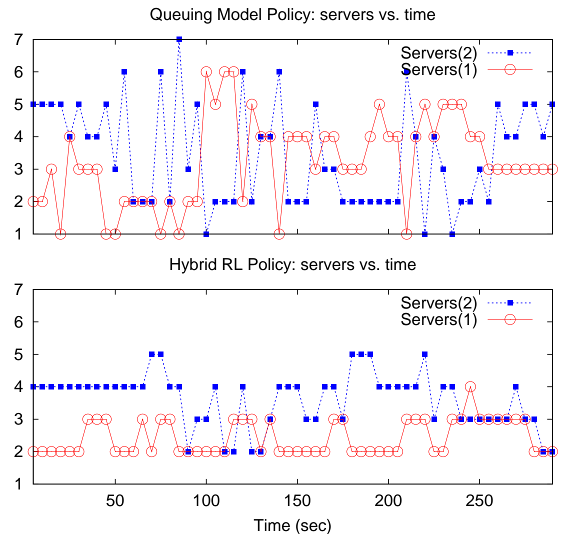

Figure 5: Reduction of thrashing using Hybrid RL in closed-loop, 4.5 sec delay experiment.

Conclusions

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 its state space or value function representation. Moreover, using 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 within a given system. By running such a policy to obtain training data, we maintain acceptable performance in the live system at all times. 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 acceptable performance while accumulating a new training set to be used for RL retraining.

Beyond server allocation, hybrid RL may have wide applicability throughout many different areas of systems management. The most promising applications 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 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 web server parameters, OS parameters, etc.. Finally, we note that hybrid RL could 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 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.

References

Bennani, M. N., and Menascé, D. A. 2005. Resource allocation for autonomic data centers using analytic performance models. In *Proc. of ICAC-05*.

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

Das, R.; Tesauro, G.; and Walsh, W. E. 2005. Model-based and model-free approaches to autonomic resource allcation. Technical Report RC23802, IBM Research.

Pradhan, P.; Tewari, R.; Sahu, S.; et al. 2002. An observation-based approach towards self-managing web servers. In *Proc. of Intl. Workshop on Quality of Service*.

Squillante, M. S.; Yao, D. D.; and Zhang, L. 1999. Internet traffic: Periodicity, tail behavior and performance implications. In Gelenbe, E., ed., *System Performance Evaluation: Methodologies and Applications*. CRC Press.

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

Tesauro, G. 2005. Online resource allocation using decompositional reinforcement learning. In *Proc. of AAAI-05*.

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

Appendix: Initial Queueing Model Policies

Our initial policies make use of standard design open-network and closed-network queuing models. We also follow recent approaches for online performance management (Pradhan *et al.* 2002; Chandra, Gong, & Shenoy 2003; Bennani & Menascé 2005) which continuously update model parameters based on dynamically varying measurements of system behavior. As per the above, we reestimate our model parameters at the end of each allocation interval, based on current measurements of mean arrival rate λ, mean response time R, number of servers allocated n, and number of customers M. The model then predicts expected response time R' in the next interval as function of various possible future number of servers n', assuming a neutral forecast of workload intensity (i.e. next intensity equals current intensity). The predicted response times in turn lead to predicted utilities $V'(n')$ as specified by the Trade3 SLA.

Open Queueing Network Model For the open-loop workload mode, we model an overall demand λ distributed among n servers using a system of n parallel identical M/M/1 queues, each with demand λ/n. This choice is dictated by the Poisson arrival process, the round-robin routing policy, and our empirical finding that service times are well approximated by an exponential distribution (apart from JVM garbage collection). Given an estimated service rate parameter μ, the M/M/1 model predicts future response time R' for n' servers according to: $R' = 1/(\mu - \lambda/n')$.

An instantaneous estimate of μ may be obtained from current measurements of λ and response time R according to: $\mu = \frac{1}{R} + \frac{\lambda}{n}$. However, such estimates are quite noisy, due to both garbage collection and finite sampling effects. To dampen these effects, we use an exponentially smoothed estimator $\tilde{\mu}$, incorporating measurements from previous intervals, in the above formula for predicted response time. Smoothing parameters in the range 0.1-0.5 empirically give the most accurate predictions, and our experiments typically set the smoothing parameter at 0.1.

Closed-Loop Queueing Network Model For the closed-loop workload mode, round-robin routing again suggests modeling an application with M customers and n servers as a system of n independent parallel closed networks, each with one server and M/n customers. We then employ the well known Mean Value Analysis (MVA) formulation to model mean response times in the individual networks. MVA requires estimating two unknown parameters: the service rate μ and the average customer think time Z. We estimate μ as in the open-loop model above, and using the Interactive Response Time Law, we estimate $Z = \frac{M}{X} - R$, where X is the measured mean overall throughput of the application. Exponential smoothing is again applied to both parameter estimates (smoothing parameter in the range 0.1-0.5) to damp out short-term fluctuations.