RC23802 (W0511-125) November 16, 2005
Computer Science

IBM Research Report

Model-Based and Model-Free Approaches to Autonomic Resource Allocation

Rajarshi Das, Gerald Tesauro, William E. Walsh
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

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

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home.

Model-Based and Model-Free Approaches to Autonomic Resource Allocation

Rajarshi Das, Gerald Tesauro and William E. Walsh

IBM TJ Watson Research Center

19 Skyline Drive, Hawthorne, NY 10532, USA

Abstract

A major goal of autonomic computing is to dynamically allocate computational resources so as to continually optimize high-level policy objectives. A key challenge to achieving this goal is to accurately estimate the impact of resource-level changes on application performance with respect to a Service Level Ageement (SLA). We compare two methodologies for accomplishing this: (i) developing a queuing-theoretic performance model for an application, and fitting its parameters online based on current state; (ii) using model-free reinforcement learning of resource valuation estimates based on trial-and-error learning. We describe these approaches in the context of a distributed architecture in which servers are allocated amongst multiple applications with independent time-varying loads. Each application has a local utility function, based on SLA payments as a function of relevant performance metrics. The overall system goal is to maximize the sum of local utility functions. Individual applications use one of the above methodologies to estimate resource valuations, which are then used by a resource arbiter to compute optimal allocations. We present empirical data illustrating the practicality and effectiveness of both methods in a realistic data center prototype. We highlight important tradeoffs between the methods, and point out potential benefits of a hybrid approach combining both methods.

1 Introduction

As today's computing systems are rapidly increasing in size, complexity and decentralization, there is now an urgent need to make many aspects of systems management more automated and less reliant on human system administrators. As a result, significant new research and development initiatives in system self-management, somtimes referred to as "autonomic computing" [6], are now under way within major IT vendors as well as academia [5]. The goals of such research include developing systems that can automatically configure themselves, detect and repair hardware and software failures, protect themselves from external attack, and optimize their performance in rapidly changing environments.

This paper addresses the important problem of dynamically allocating resources in a distributed computing system responsible for handling many time-varying workloads. Internet data centers, which often utilize hundreds of servers to service dozens of high-volume web applications, provide a prime example where dynamic resource allocation may be extremely valuable. High variability in load for typical web applications implies that, if they are statically provisioned to handle their maximum possible load, the average utilization ends up being low, and resources are used inefficiently. By dynamically reassigning servers to applications where they are most valued, resource usage can be much more efficient. This problem has been addressed in both research efforts [1, 10] and commercial software [15].

According to general principles of autonomic computing, resource allocation decisions should be governed by a suitable high-level policy objective. In this paper, we assume that there is a precisely defined local utility function for each application, based on relevant local performance metrics, and that the overall system objective is to maximize the sum of local utility functions. We outline the approach below; a

detailed rationale for it can be found in [16].

Our primary aim in this paper is to compare and contrast two radically different methodologies for making server allocation decisions. The *model-based* approach uses an appropriate queuing-theoretic performance model to estimate how changes in allocated servers affect expected performance of an application. The *model-free* approach uses reinforcement learning to directly learn how to estimate expected utility of an application, given the application's current state and number of servers allocated. This approach, which does not require an explicit system performance model, is original to our knowledge. The substantial differences between these methods make for a number of interesting comparisons and tradeoffs, which we examine in addition to a direct head-to-head comparison of how well each method performs in optimizing systemwide utility.

The rest of this paper is organized as follows. Section 2 reviews our distributed data center architecture and implemented prototype, comprising real servers and realistic Web-based traffic, in which we implement and test our two approaches. Section 3 describes the queuing model approach, while Section 4 describes the reinforcement learning approach. Section 5 presents a detailed comparison of the two approaches in the data center prototype. Section 6 summarizes our contributions and discusses ongoing and future extensions of the work presented here.

2 Data Center

2.1 Data Center Architecture

The data center architecture [16], illustrated in Figure 1, contains a number of logically separated *Application Environments*, each providing a distinct application ser-

vice using a dedicated, but dynamically allocated, pool of servers. Each Application Environment has a *service-level utility function* specifying the value of providing a given level of service to users of the Application Environment. The utility function will typically reflect the payment/penalty terms of service-level agreements with customers, but may also incorporate additional considerations such as the value of maintaining the data center's reputation for providing good service. We assume each Application Environment's utility function is independent of that of other Application Environments, and that all utility functions share a common scale of valuation, such as money. The utility function for environment i is of the form $U_i(\mathbf{T}_i)$, where $\mathbf{T}_i$ is a vector of attributes for i. While, in general, the argument $\mathbf{T}_i$ can involve any number metrics of interest to the Application Environment users, in the present work we assume in this work that $\mathbf{T}_i = T_i$ indicates a single metric for a single service class, although that metric need not be the same for all Application Environments.

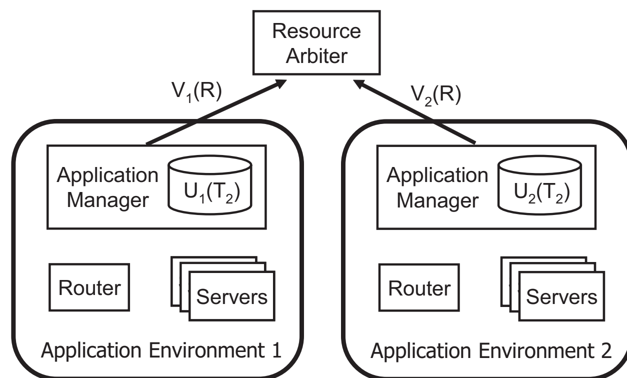

Fig. 1. Data center architecture.

The system goal is to optimize $\sum_i U_i(T_i)$ on a continual basis to accommodate fluctuations in demand. The global optimization task is distributed in a two-level structure. At the lower level, the detailed control within an Application Environment, such as routing and control parameter tuning, is handled by an associated *Application Manager*. At the higher level, a global *Resource Arbiter* allocates resources

across different Application Environments. The detailed state information of an Application Environment is known only by its Application Manager, but not to the Arbiter. Instead, when server reallocation is considered, an Application Manager sends to the Arbiter a *resource-level utility function* $V_i(R_i)$ that specifies the value to the Application Environment of obtaining each possible number R_i of servers.

Given the current functions $V_i(R_i)$ from the Application Managers, the Arbiter periodically recomputes the resource allocation R^* that maximizes the global utility $\sum_i U_i(T_i) = \sum_i V_i(R_i)$: $R^* = \arg\max_R \sum_i V_i(R_i)$ s.t. $\sum_i R_i = \bar{R}$, where $\bar{R}$ indicates the total number of servers available. Each server is allocated as a whole unit and cannot be be shared among Application Environments.

2.2 Data Center Prototype Implementation

With our colleagues [2], we implemented a prototype of our data center in a general software architecture for autonomic computing systems called Unity. It is implemented on a cluster of identical IBM eServer xSeries 335 machines running Redhat Enterprise Linux Advanced Server. We run two kinds of applications in Unity, both of which run in separate Application Environments and are installed on all servers. As such, there is negligible delay in switching servers between Application Environments.

The "Trade3" application [4] is a realistic simulation of an electronic trading platform, designed to benchmark web servers. This transactional workload runs on top of IBM WebSphere and DB2. For the Trade3 utility function, T_i is average transaction response time. Demand in Trade3, measured in units of page requests per second, is simulated using an open-loop Poisson HTTP request generator with an adjustable mean arrival rate. To provide a realistic emulation of stochastic bursty

time-varying demand, a time series model of Web traffic developed by Squillante et al. [12] is used to reset the mean arrival rate every 2.5 seconds. The transaction requests to Trade3 are distributed round-robin among its servers.

The "Batch" application handles a long-running, parallelizable batch workload that can be paused and restarted on separate servers as they are added and removed. The service metric is throughput, but, since servers are homogeneous and there is no notion of time-varying demand, we equivalently define utility directly as a function of servers. That is, $V_i(R_i) = U_i(T_i)$.

2.3 Computing Resource-level Utility Functions

The core challenge we address in this paper is for the Application Manager to dynamically compute $V_i(R_i)$ for a transactional workload in response to changing demand and relevant internal state. In a *model-based approach*, an Application Manager i has a model $T_i(R_i, S_i)$ that indicates the performance its Application Environment can achieve in state S_i if it has R_i number of servers. The relevant state may be composed of any measurable quantities observed thus far, including demand, queue length, response time, and utilization. The model may include predictions about how any exogenous state components, such as demand, may evolve. With the model, the Application Managers computes $V_i(R_i) = U(T_i(R_i, S_i))$. In the *model-free approach*, the Application Managers uses a learned value function $Q_i(R_i, S_i)$ to compute its resource-level utility function directly from the resources and Application Environment state without an intervening model: $V_i(R_i) = Q_i(R_i, S_i)$. Although not modeled explicitly, the process for learning $Q_i(R_i, S_i)$ is designed to reflect the service-level utility that would be obtained with R_i, given the evidence provided in the state. Note that, with both methods, the Application Managers reports V_i only as a function of R_i because the Arbiter need not know S_i. All computation involv-

ing the state is handled inside the Application Managers. We discuss two particular instances of these general methods in greater detail in the following sections.

When designing and evaluating an approach to computing resource-level utility functions, we focus primarily on three issues: 1) *Performance*. Typically, the measure of performance for a model-based approach is the accuracy of the model. For the model-free approach the closest correspondence would be accuracy of the resulting resource-level utility function. However, in both cases, the true values in a dynamic, real system can be elusive. Nevertheless, what we ultimately care about is not accuracy per se, but the level of global utility obtained in the system, hence we use this as our measure of performance. 2) *Design costs*. Ultimately, the goal of autonomic computing is to reduce the amount of administrative effort required to configure, optimize, and maintain a system. Clearly then, we want to avoid the need of Ph.D.-level expertise every time a system is set up, reconfigured, or usage requirements are changed. 3) *Transient runtime costs*. Some model-based and model-free approaches may involve learning or other adaptation. It is important that this does not incur unacceptably long transient periods of low performance.

3 Queuing Model Approach

There has been significant recent interest on efficient resource allocation in large data centers in order to provide guarantees on resource availability and performance for multiple enterprise applications. Most, if not all, of the published work in this area use simple to sophisticated queuing models—based on the richness of state information obtained through online measurements—to model the performance of the computational resources in the data center [1, 3, 7, 8, 9, 10]. Except in [7], the overall system goal in these works is to minimize the quality of service deviations defined unilaterally by the data center. On the other hand, in a spirit similar to

our current work, the methodology in [7] attempts to maximize revenues that are generated by satisfying quality of service guarantees derived from flexible service level agreements between service providers and their clients. However, unlike our work described here, the methodology for maximizing revenues is studied solely via numerical simulations.

Here we discuss one specific approach for an Application Manager handling Trade3 to obtain a resource model and the corresponding resource-level utility function. To account for the transient nature of the workload as well as the variance in the performance of servers due to garbage collection and thread management in Java, the estimated parameters of the model are updated at periodic intervals. In each allocation period t, the Application Manager is cognizant of the following state information: (i) the demand $\lambda^t = D^t$, denoting the mean arrival rate of requests (assuming a single service class), (ii) τ^t, the mean end-to-end response time of the completed requests, and (iii) $R^t > 0$, the number of allocated server machines. Since the arriving requests are allocated among the available servers, each with its own queue managed by WebSphere in a round-robin fashion, we use R^t parallel M/M/1 queues to model the behavior of the resources at time interval t. Thus, the estimated mean service rate of a single server, μ^t, is given by [1]

$$\mu^t = \frac{1}{\tau^t} + \frac{\lambda^t}{R^t}. \tag{1}$$

The above estimate of μ^t is sensitive to large fluctuations in server machine performance. To damp corresponding potential short-term fluctuations in μ^t, we use instead a smoothed quantity $\hat{\mu}^t$, obtained by exponential smoothing of μ^t with parameter $\omega = 0.5$ as follows: $\hat{\mu}^t = \omega\, \hat{\mu}^{t-1} + (1-\omega)\, \mu^t$.

Given $\hat{\mu}^t$ and the predicted demand $\tilde{\lambda}^{t+1}$, the Application Manager can predict the

[1] For convenience, we drop the index i for the Application Environment/Manager.

mean response time for a proposed new allocation of servers R^{t+1} in the next period $t+1$. Given the slowly varying nature of the demand model, in our current implementation we approximate $\tilde{\lambda}^{t+1} \cong \lambda^t$, and thus, after simple rearrangement of terms, we obtain

$$\hat{\tau}^{t+1} = \frac{1}{\hat{\mu}^t - \frac{\lambda^t}{R^{t+1}}}. \tag{2}$$

In allocation period t, the Application Manager uses Equation 2 to compute $V(R^t) = U(\tau^t)$ for each possible number of servers R^t.

The queuing model employed in our work is relatively simple, yet works effectively in our prototype system. However, large, deployed, industrial data centers are complex, reconfigurable, multitiered systems with dozens of components and hundreds of tunable parameters. More complex models may be required to obtain good performance in such systems, but increased model complexity comes with a potential cost. System administrators typically do not have the necessary expertise to formulate appropriate complex models. Alternate complex models could be made available in software packages, leaving administrators to make frequent critical choices between models while simultaneously configuring the relevant parameters. Since this runs counter to the autonomic computing goal of making computational systems less reliant on humans, there must be some tradeoff between the complexity and accuracy of available models.

4 Reinforcement Learning Approach

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 in the state, 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. RL methods have solid theoretical grounding for Markov Decision Problems (MDPs), and in the last decade there have been many notable success stories in a variety of real-world applications. An excellent general overview of RL is given in [13].

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 [13]. (For simplicity we hereafter omit the π subscript.) The Sarsa rule has the following form:

$$\Delta Q(s^t,a^t) = \alpha(t)[r^t + \gamma Q(s^{t+1},a^{t+1}) - Q(s^t,a^t)] \quad (3)$$

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. Note that when a lookup table is used to represent the values of state-action pairs (which we do here), then each state-action pair has to be sampled many times to accurately learn its value. This usually entails some form of "exploration" rule ensuring visits to all state-action pairs, even those thought by the RL agent to yield low expected value.

Equation 3 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. While our applications may treated approximately as MDPs, the above policy conditions do not strictly hold in our system. Taking s to be the application's local state, and a to be the local

allocation decision of the arbiter, then the arbiter's policy is neither stationary nor locally greedy. Instead, the arbiter's task may be described formally as a composite MDP [11], and its policy of maximizing the sum of all value functions is neither stationary nor greedy from a purely local perspective. The issue of whether local RL converges in composite MDPs in general is an interesting open research topic which we address here empirically, and which is discussed in more detail in [14]. Additional issues that we face is using RL in our system are detailed below.

4.1 Important Practical Issues for RL

The main issues for practical success with RL are generally: (i) avoiding excessively long training times; (ii) avoiding excessive penalties for poor performance (including penalties for exploring suboptimal actions) during training; (iii) dealing with potentially non-Markovian system effects. We address each of these below.

One of the most crucial practical issues is the design of a good state-space representation scheme. Potentially many different types of sensor readings (e.g. average demand, response time, CPU and memory utilization, number of Java threads running, etc.) may be needed to accurately describe the system state. Additional historical information may need to be maintained if there are important history-dependent effects (e.g. Java garbage collection). Encoding of the state description needs to be appropriate to whatever value function approximation scheme is used and must be sufficiently rich to capture the underlying function accurately yet sufficiently compact if a lookup table is used (so that the table can be effectively explored in a reasonable amount of time).

Another important issue arises from the physical time scales associated with live training in a real system. These are often much slower than simulation time scales,

so that learning needs to be much faster (in terms of number of value function updates) relative to training times that are often acceptable in simulation. This can be addressed in part by using a sufficiently compact value function representation. We also advocate, as described in more detail below, the use of heuristics or domain knowledge to define good initial states for value function training; this can make it much easier and faster for RL to find the asymptotic optimal value function. In addition, we also advocate "hybrid" training methods in which an externally supplied policy (coming from, for example, model-based methods) is used during the early phases of learning. In our system hybrid training appears to hold the potential, either alone or in conjunction with heuristic initialization, to speed up training by an order of magnitude relative to standard techniques.

A final and perhaps paramount issue for live training is that one cares about not only asymptotic performance, but also rewards obtained during learning. This may be unacceptably low due to both exploration of alternative actions and a poor initial policy. The latter factor can be addressed by clever initialization and hybrid training as mentioned above. In addition, we expect that some form of safeguard mechanism and/or intelligent exploration will be needed to limit penalties incurred during exploration. Perhaps surprisingly, this was unnecessary in our prototype system: a simple rule of choosing a random action, rather than the action currently estimated to be optimal, with a probability of 0.1 incurs negligible loss of expected utility. Nevertheless, we expect that intelligent/safe exploration would eventually become necessary as our system increases in complexity and realism.

4.2 RL Implementation Details

The operation of RL within the Trade3 application manager is shown in Figure 2. The RL module learns a value function expressing the long-range expected value

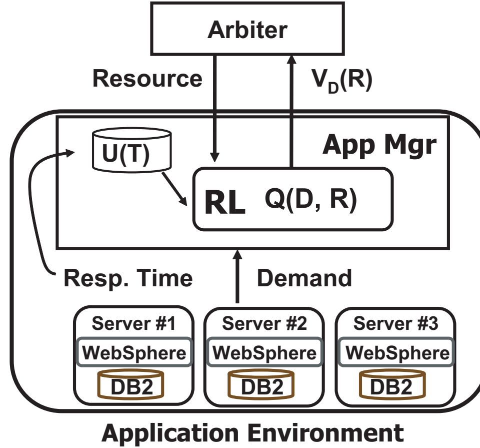

Fig. 2. Operation of RL in the Application Manager.

associated with a given current state of the workload and current number of servers assigned by the arbiter. (We stress that policy decisions are made externally and are not controlled by the RL agent, unlike in normal usage of RL.) RL runs as an independent process inside the Application Manager, and uses its own clock to generate discrete time steps every 2.5 seconds. Thus the RL time steps are not synchronized with the arbiter's allocation decisions.

While there are many sensor readings that could be used in the workload state description, for simplicity we use only the average demand D in the most recent time interval. Hence the value function $Q(D,R)$ is a two-dimensional function, which is represented as a two-dimensional grid. The continuous variable D has an observed range $\sim 0-325$ (in units of page requests per second), which is discretized into ~ 130 grid points with an interval size of 2.5. The number of servers R is an integer between 1 and 5 (by fiat we prohibit the arbiter from assigning zero servers to Trade3), so the total size of the value function table is about 650. Note that our RL implementation uses less state information than does the queuing model. This yields a smaller grid for Q, which learns faster but is still effective for resource allocation, as we describe in Section 5.

At each time step the RL module observes the demand D, the resource level R given by the arbiter, and utility U (which, recall, is a function of average response time T). It then updates its value table using Sarsa(0), with discount parameter $\gamma = 0.5$ and, for cell i, learning rate $\alpha(t) = 0.2c/(c+v_i)$. The $c/(c+v_i)$ factor is a standard decay of the learning rate, where v_i is the number of visits to cell i and $c = 80$ is constant.

It is important to note that the distribution of cell visits is highly nonuniform, due to nonuniformity in both demand and resource allocations. To improve the valuations of infrequently visited cells, and to allow generalization across different cells, we use soft monotonicity constraints that encourage cell values to be monotone increasing in R and monotone decreasing in D. Specifically, when RL updates a cell's value, if this update violates monotonicity with any other cells in its row or column, the violation is partly repaired, with greater weight applied to the less-visited cell. Monotonicity with respect to servers is a very reasonable assumption for this system. Monotonicity in demand also seems to be reasonable in our system (although for certain dynamical patterns of demand variation, it may not strictly hold). Both constraints yield significantly faster and more accurate learning.

5 Results

This section presents typical results obtained using the queuing model and RL approaches in our system. We first briefly discuss empirical evidence for convergence of RL to a stationary value function. We then present a head-to-head comparison of the two approaches in terms of optimizing total system utility in a two-application scenario. We then examine scalability of both approaches using a more complex scenario with three applications.

In our experiments, the service-level utility for Trade3 is a decreasing function of average response time over a five second interval. The equation is $200/(1 + e^{(x-40)/5}) - 100$, which is a sigmoid centered around 40ms. The Batch utility utility function assigns values $(-20, 8, 34, 58, 80, 100)$ directly to zero through five servers. The Arbiter requests resource utility curves from each Application Manager every five seconds, and thereupon computes a new allocation.

5.1 Empirical Convergence of RL

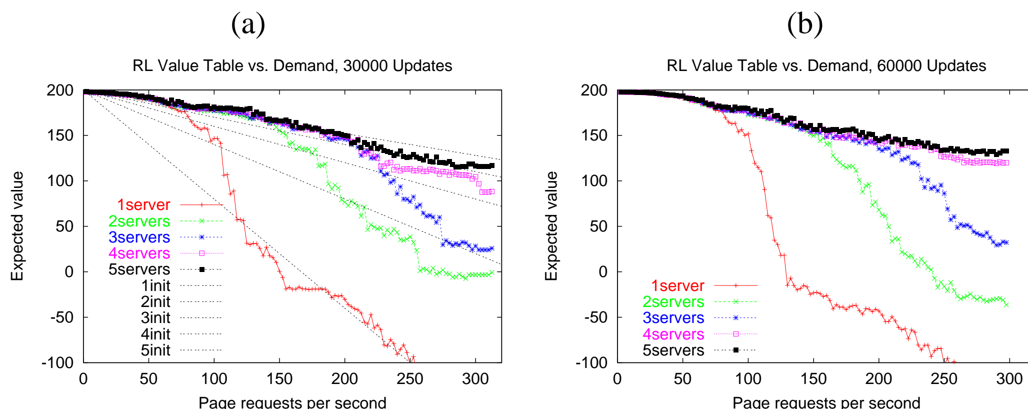

Fig. 3. (a) Trade3 value function trained from heuristic initial condition, indicated by straight dashed line. (b) Continuation of training from (a), with visit counts reset to zero.

Figure 3(a) shows RL results in an overnight run in our standard two-application scenario (Trade3 + Batch) with five servers, using a simple but intelligently chosen heuristic initial condition. The heuristic is based on the intuitive notion that the expected performance and hence value in an application should depend on the demand per server, D/R, and should decrease as D/R increases. A simple guess is linear dependence, and the five straight lines are the initial conditions $Q_0 = 200 - 1.2(D/R)$ for $R = 1$ to 5. The overall performance during this entire run is high, even including the 10% random arbiter exploration.

Figure 3(b) shows results when training is continued for an additional overnight

session with visits counts reset to zero so that the learning rates would again start from maximal values. We see relatively little change in the learned value function, suggesting that it may be close to the ideal solution. (One can't be sure of this, however, as the number of visits to non-greedy cells is quite small.) Notably, in a comparable amount of training starting from random initial conditions, a solution is found that is close to Figure 3(b), providing additional evidence of RL convergence, as similar solutions were found starting from radically different initial conditions.

5.2 Performance Results

Figure 4 compares the performance of RL (specifically the run shown in Figure 3(a)) and queuing model approaches in the standard two-application scenario, using identical demand generation in each run. Performance is measured over the entire run in terms of average total system utility earned per arbiter allocation decision. To establish a range of possible performance values, we also compare with two inferior allocation strategies: "UniRand" consists of uniform random arbiter allocations; and "Static" denotes the best static allocation (three servers to Trade3 and two to Batch). Also shown is a dashed line indicating an analytical upper bound on the best possible performance that can be obtained in this system using the observable information.

The analytic upper bound of the best possible performance was determined through expected utility analysis. Using the experimentally derived mean service rate of $0 < R \leq \bar{R}$ servers and the probability distribution of the demand, we find the joint probability distribution over the demand and the number of allocated servers that results in maximum mean expected system utility.

Note that the RL performance includes all learning and exploration penalties, which

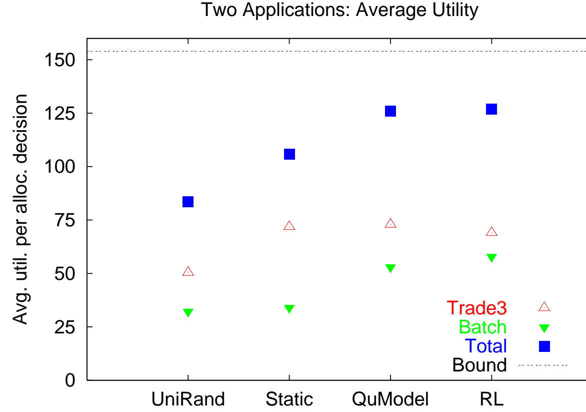

Fig. 4. Performance of various strategies in scenario with two applications.

are not incurred in the other approaches. Both the model-based and model-free approaches are substantially better than static or random allocations, and they are also reasonably close to the maximum performance bound, which is a bit generous and most likely can be tightened to lie below the 150 level. We also find it surprising that, despite their radically different natures, the performance figures for RL-based and queuing-model-based allocations are virtually identical. In fact, if one examines the individual allocation decisions at a given demand level, we find that they are usually identical. The only slight difference we can detect is that the queuing-model-based allocator tends to be a bit more conservative, preferring to assign larger number of servers to Trade3, whereas RL is a bit more aggressively optimistic that Trade3 can obtain high utility with slightly fewer servers.

5.3 Scaling to Additional Applications

We have also compared our approaches in a more complex scenario containing three applications: one Batch plus two separate Trade3 environments, each with an independent demand model, shown in Figure 5(a). While this should have no effect on the performance of individual queuing models, from an RL perspective this scenario is more challenging in that there are now multiple interacting RL mod-

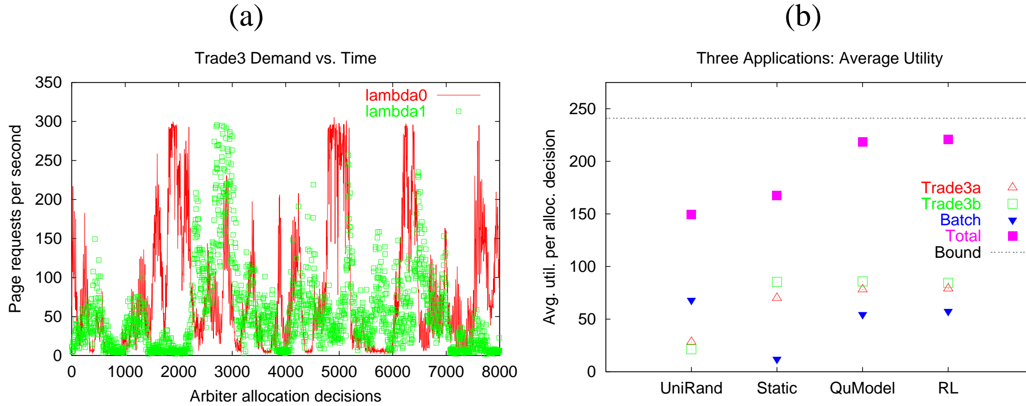

Fig. 5. (a) Independent time-varying demand models in two Trade3 environments used in scenario with three applications. (b) Performance of various strategies in scenario with three applications.

ules, each of which induces non-stationarity in the other's environment. However, using the same heuristic demand per server initialization as before, we observe no qualitative difference in RL training times, apparent convergence, or quality of policy compared to previous results. Performance results in this scenario are plotted in Figure 5(b). Once again, RL performance is comparable to the queuing model approach (about 1% better, in fact, although we cannot claim this difference is significant), and both are quite close to the maximum possible performance. (That performance is closer to the upper bound than with two applications could be attributable to a tighter upper bound, better performance, or both.) Once again RL and the queuing model are almost always in agreement on specific allocation decisions. While this does not establish scalability to an arbitrary number of applications, the results are encouraging regarding our general methodology. They also suggest that RL may well be a viable alternative to model-based approaches for real-world problems of this type.

6 Conclusions

We have presented and demonstrated effectiveness of two substantially different approaches to estimating resource valuation within an overall framework for online allocation of computational resources amongst multiple applications with independent time-varying need for resource.

The more established queuing-theoretic model-based approach has a venerable record of success in capacity planning based on offline estimation of queuing model parameters. More recently there have also been several success stories using these models more dynamically via online parameter estimation. A common rule of thumb amongst practitioners is that queuing models can predict system behavior to within 20%, which is often more than adequate for many practical purposes.

The novel model-free reinforcement learning approach has no prior track record in systems management, but could have a bright future in store. Based on experience in other domains, we initially expected that in real computing systems, RL might well require exorbitant training times, complex state descriptions and nonlinear function approximation techniques, and suffer from extremely poor performance during training. However, the experience with RL in our prototype system has turned out better than expected in all of these respects. We find that with extremely simple state descriptions (average demand only), value function representations (a uniform grid), exploration schemes (10% random allocations), and initialization schemes (linear in demand per server), we achieve performance comparable to our best efforts within the model-based framework. Moreover, these results were obtained in an eminently feasible amount of training time.

Whether such success with RL will continue as we examine progressively more complex systems is of course an interesting open research question. We fully expect

that nonlinear function approximators, such as neural networks or support vector machines, will eventually be required to represent RL value functions. This will open up a range of new challenges that we have not faced in current work. We also expect to need explicit techniques for obtaining acceptable performance levels during training, and for intelligent exploration of non-greedy actions. A promising candidate for this is Boltzmann exploration, in which the probability of action selection is based on current valuation estimates. Another particularly promising technique, which we have mentioned previously, is hybrid training, in which the policy decisions in the initial phases of learning are model-based. This can provide not only a strong initial policy, but can also provide safety bounds on exploration, if one can establish that exploring actions close to the model-based recommendation will yield performance close to the model-based performance.

Apart from raw performance comparisons, we identify at least two additional important aspects for future practical comparisons of model-based and model-free methods. One is the amount of systems knowledge required for success with each method. RL appears to have the initial edge in this regard, and it will be interesting to see if this persists in subsequent work. Secondly, there is the issue of brittleness of performance models and learned value functions under various forms of environmental or system changes. Certainly some forms of system change, such as changing the service-level utility function, will not require any queuing model changes, whereas RL might need to be retrained from scratch. On the other hand, certain changes in user behavior may lead to gradual "model drift" in which the queuing model progressively becomes less accurate. If this drift is slow, RL might be able to continually adapt the value estimates so that they maintain their accuracy as conditions change.

In ongoing and future work, we will address effects arising when there are sig-

nificant delays in reassigning servers to different workloads. These are negligible in the current system. This appears to require innovation within the model-based framework. However, as RL is intrinsically designed to handle situations involving delayed reward, we expect it to continue to perform well without significant modifications. We also plan to investigate allocation of different types of computational resources, such as storage devices or database access. Finally, we aim to further develop our server allocation techniques to the point where they are demonstrably deployable within commercial server management systems.

References

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

[2] D. Chess, A. Segal, I. Whalley, and S. White. Unity: Experiences with a prototype autonomic computing system. In *1st IEEE International Conference on Autonomic Computing*, pages 140–147, 2004.

[3] R. Doyle, J. Chase, O. Asad, W. Jen, and A. Vahdat. Model-based resource provisioning in a web service utility. In *USENIX Symposium on Internet Technologies and Systems*, 2003.

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

[5] ICAC. *Proceedings of the First International Conference on Autonomic Computing*. IEEE Computer Society, Los Alamitos, CA, 2004.

[6] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–52, 2003.

[7] Z. Liu, M. Squillante, and J. L. Wolf. On maximizing Service-Level-Agreements profits. In *Proceedings of Electronic Commerce Conference*. ACM, 2001.

[8] D. Menasce, V. Almedia, and L. Dowdy. *Performance by design: Computer Capacity*

Planning by Example. Prentice Hall, Upper Saddle River, NJ, 2004.

[9] D. Menasce, D. Barbara, and R. Dodge. Preserving QoS of e-commerce sites through self-tuning, a performance model approach. In *Proceedings of Electronic Commerce Conference*. ACM, 2001.

[10] S. Ranjan, J. Rolia, E. Knightly, and H. Fu. QoS-driven server migration for internet data centers. In *Proceedings of ACM/IEEE Intl Workshop on Quality of Service (IWQoS)*, 2002.

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

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

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

[14] G. Tesauro. Decompositional reinforcement learning and workload management. Submitted for publication, 2005.

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

[16] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das. Utility functions in autonomic systems. In *1st IEEE International Conference on Autonomic Computing*, pages 70–77, 2004.