

IBM Research Report

Online Resource Allocation Using Decompositional Reinforcement Learning

Gerald Tesauro
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

Online Resource Allocation Using Decompositional Reinforcement Learning

Gerald Tesauro

IBM TJ Watson Research Center
19 Skyline Drive, Hawthorne, NY 10532, USA
gtesauro@us.ibm.com

Abstract

This paper considers a novel application domain for reinforcement learning: that of “autonomic computing,” i.e. self-managing computing systems. RL is applied to an online resource allocation task in a distributed multi-application computing environment with independent time-varying load in each application. The task is to allocate servers in real time so as to maximize the sum of performance-based expected utility in each application. This task may be treated as a composite MDP, and to exploit the problem structure, a simple localized RL approach is proposed, with better scalability than previous approaches. The RL approach is tested in a realistic prototype data center comprising real servers, real HTTP requests, and realistic time-varying demand. This domain poses a number of major challenges associated with live training in a real system, including: the need for rapid training, exploration that avoids excessive penalties, and handling complex, potentially non-Markovian system effects. The early results are encouraging: in overnight training, RL performs as well as or slightly better than heavily researched model-based approaches derived from queuing theory.

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 this area, commonly referred to as “Autonomic Computing” (Kephart & Chess 2003), are now under way within major IT vendors as well as throughout academia (ICAC 2004). 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 considers the use of reinforcement learning (RL) techniques for the important problem of optimal online allocation of resources. Resource allocation problems are pervasive in many domains ranging from telecommunications to manufacturing to military campaign planning. In autonomic computing, resource allocation problems are

faced in large-scale distributed computing systems responsible for handling many time-varying workloads. Internet data centers, which often utilize hundreds of servers to handle 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.

The standard approach to resource allocation in computing systems entails developing a system performance model using an appropriate queuing-theoretic discipline. This often requires a detailed understanding of the system design and the patterns of user behavior. The model then predicts how changes in resource affect expected performance. Such models perform well in many deployed systems, but are often difficult and knowledge-intensive to develop, brittle to various environment and systems changes, and limited in capacity to deal with non-steady-state phenomena.

Reinforcement learning, by contrast, offers the potential to develop optimal allocation policies without needing explicit system models. RL can in principle deal with non-trivial dynamical consequences of actions, and moreover, RL-based policies can continually adapt as the environment changes. Hence RL may be naturally suited to sequential resource allocation tasks, provided that: (1) the effects within an application of allocating or removing resources are approximately Markovian, given a suitable state representation; (2) there is a well-defined reward signal for each application that RL can use as its training signal. In this case, the proper objective of sequential resource allocation, finding the optimal allocation policy that maximizes the sum of cumulative discounted reward for each application, is well-suited to RL methods.

Note that in many computational resource allocation scenarios, a current allocation decision may have delayed consequences in terms of both expected future rewards and expected future states of the application. Such effects can arise in many ways, for example, if there are significant switching delays in removing resources from one application and reassigning them to a different application. Despite the complexity of such effects, RL should in principle be able to deal

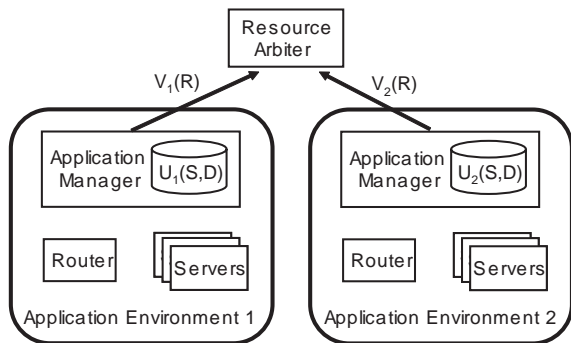


Figure 1: Data center architecture.

with them, as they are native to its underlying formalism. On the other hand, such delayed effects are more difficult to treat in other frameworks such as steady-state queuing theory.

Resource Allocation Scenario

We begin by describing a prototype Data Center scenario for studying resource allocation, which generally follows the approach outlined in (Walsh *et al.* 2004). As illustrated in Figure 1, the Data Center prototype contains a number of distinct Application Environments, each of which has some variable number of dedicated servers at its disposal for processing workload. In this scenario, each application has its own *service-level utility function* $U_i(S, D)$ expressing the value to the Data Center of delivering service level S to the application’s customers at demand level D . This utility function, which we use as the RL reward signal, will typically be based on payments and penalties as specified in a Service Level Agreement (SLA), but may also reflect other customer relationship considerations such as the Data Center’s reputation for delivering consistently good service. The service level S may consist of one or more performance metrics relevant to the application (e.g. average response time or throughput). We assume that each application’s U_i is independent of the state of other applications, and that all U_i functions share a common scale of value, such as dollars.

The decision to reallocate servers amongst applications is periodically made by a Resource Arbiter, on the basis of resource utility curves $V_i(R_i)$ received from each application. These express the expected value of receiving R_i servers, and ultimately derive from the corresponding service-level utility functions. The Arbiter’s allocation decision maximizes the sum of resource utilities: $R^* = \arg \max_R \sum_i V_i(R_i)$, where the feasible joint allocations cannot exceed the total number of servers within the Data Center. This can be computed for A applications and S homogeneous servers in time $\sim O(AS^2)$ using a simple Dynamic Programming algorithm.

Prototype System Details

A typical experiment in the Data Center prototype uses two or three distinct Application Environments, five servers devoted to handling workloads, plus an additional server to run the system management elements (Arbiter, Application Managers, etc.). The servers are IBM eServer xSeries machines running RedHat Enterprise Linux.

Two types of workloads are implemented in the system. One is “Trade3” (2004), a standard Web-based transactional workload providing a realistic emulation of online trading. In the Trade3 simulation a user can bring up Web pages for getting stock quotes, submitting buy and sell orders, checking current portfolio value, etc.. In order to run Trade3, the servers make use of WebSphere and DB2¹. User demand in Trade3 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 times series model of Web traffic developed by Squillante *et al.* (1999) is used to reset the mean arrival rate every 2.5 seconds. (An illustration of the model’s behavior is given later in Figure 5.) The other workload is a CPU-intensive, non Web-based “Batch” workload meant to represent a long-running, offline computation such as Monte Carlo portfolio simulation.

The SLA for Trade3 specifies payment as a function of average response time over a five second interval in terms of a sigmoidal function centered at 40 msec with width ~ 20 msec. High payments are received for short response times, while very long response times receive a negative penalty. The Batch SLA is a simple increasing function of the number of servers currently assigned. The relative amplitudes of the two SLAs are chosen so that Trade3 is somewhat more important, but not overwhelmingly so. The Arbiter requests resource utility curves from each application every five seconds, and thereupon recomputes its allocation decision.

Decompositional RL Approach

We now consider ways to use RL in this scenario. Since the impact of an allocation decision is presumed to be Markovian within each application, we can regard the Arbiter’s problem as a type of MDP, and can implement a standard RL procedure such as Q-learning within the Arbiter. The state space for this MDP is the cross product of the local application state spaces, the action space is the set of feasible joint allocations, and the reward function is the sum of local application rewards. Such an approach, while presumably correct, fails to exploit problem structure and should scale poorly, since the state and action spaces scale exponentially in the number of applications. This should lead to curse of dimensionality problems when using standard lookup table Q-functions. While this can be addressed using function approximation or sample-based direct policy methods, one nonetheless expects the problem to become progressively more difficult as the number of applications increases.

An alternative approach is to use a decompositional formulation of RL that exploits the structure of the Arbiter’s problem. This general topic is of considerable recent interest, and structured RL approaches have been studied, for example, in hierarchical (Dietterich 2000) and factored (Kearns & Koller 1999) MDPs. The scenario here may be characterized formally as a *composite MDP* (Singh & Cohn 1998; Meuleau *et al.* 1998), i.e. a set of N independent MDPs that are coupled through a global decision

¹WebSphere and DB2 are IBM software platforms for managing Web applications and databases, respectively.

maker, with constraints on the allowable joint actions (otherwise each MDP can be solved independently).

One recent approach we might consider is Q-decomposition of Russell and Zimdars (2003), in which subagents learn local value functions expressing cumulative expected local reward. While the reward is decomposed in this approach, the state and action spaces are not: each learning agent observes the full global state and global action. Hence this approach again faces exponential scaling in the number of agents. Another approach is Coordinated Reinforcement Learning of Guestrin et al. (2002), which learns local value functions by exploiting a coordination graph structure. This may be of interest when there is no global decision maker. However, there are no local rewards in this approach, and it does not appear well-suited for resource allocation problems as the requisite coordination graph is fully connected, and each agent potentially must observe the actions and states of all other agents.

In contrast to the above approaches, this paper proposes a simple but apparently novel fully decompositional formulation of RL for composite MDPs. Each Application Manager i uses a localized version of the Sarsa(0) algorithm to learn a local value function Q_i according to:

$$\Delta Q_i(s_i, a_i) = \alpha[U_i + \gamma Q_i(s'_i, a'_i) - Q_i(s_i, a_i)], \quad (1)$$

where s_i is the application’s local state, a_i is the local resource allocated by the Arbiter, and U_i is the local SLA-specified reward. When the Arbiter requests a resource utility curve, the application then reports $V_i(a_i) = Q_i(s_i, a_i)$. The Arbiter then chooses a feasible joint action maximizing the sum of current local estimates, i.e., $a^* = \arg \max_a \sum V_i(a_i)$. As in all RL procedures, the Arbiter must also perform a certain amount of non-greedy exploration, which is detailed below. We note, along with Russell and Zimdars, that an on-policy rule such as Sarsa is needed for local learning. Q-learning cannot be done within an application, as its learned value function corresponds to a locally optimal policy, whereas the Arbiter’s globally optimal policy does not necessarily optimize any individual application.

Scaling and Convergence

Perhaps surprisingly, purely local RL approaches have not been previously studied for composite MDPs. Certainly the scalability advantages of such an approach are readily apparent: since local learning within an application does not observe other applications, the complexity of local learning is independent of the number of applications. This should provide sufficient impetus to at least empirically test local RL in various practical problems to see if it yields decent approximate solutions. It may also be worthwhile to investigate whether such behavior can be given a sound theoretical basis, in at least certain special-case composite MDPs.

With the procedure described above, we obtain favorable scaling possibly at the cost of optimality guarantees: the procedure unfortunately lies beyond the scope of standard RL convergence proofs. One can argue that, if the learning rates in each application go to zero asymptotically, then all local value functions will become stationary, and thus the global policy will asymptotically approach some stationary policy.

However, this need not be an optimal policy for the global MDP. Russell and Zimdars argue that their local Sarsa learners will correctly converge under any fixed global policy. However, this depends critically on the local learners being able to observe global states and actions. From the perspective of a learner observing only local states and actions, a fixed global policy need not appear locally stationary, so the same argument cannot be used here.

A simple counter-example suffices to establish the above point. Consider two local applications, A and B , that are provided resources by the global decision maker. Suppose that A ’s need for resource is governed by a stationary stochastic process, while B ’s has a simple daily periodicity, e.g., B is a stock-trading platform that needs many resources while the market is open, and minimal resources otherwise. Unless A has time of day in its state description, it will fail to capture the temporal predictability of the global policy, and thus the value function that it learns with local RL will be suboptimal relative to the function that it could learn were it provided with the additional state information.

The extent to which local state descriptions may be provably separable without impacting the learning of local value functions is an interesting issue for future research in decompositional RL. Most likely the answer will be highly domain-specific. One can plausibly envision many problems where the impact of local MDPs on global policy is a stationary i.i.d. process. Also, their temporal impact may be learnable by other MDPs if they are intrinsically correlated, or by using simple state features such as time of day. In other problems, no effective decomposability may be attainable. In the remainder of the paper, we consider convergence as a matter to be examined empirically within the specific application. We also adopt the practitioner’s perspective that as long as RL finds a “good enough” policy in the given application, it may constitute a practical success, even though it may not find the globally optimal policy.

Important Practical Issues for RL

Many significant practical issues need to be addressed in applying RL to the complex distributed computing system described above. One of the most crucial issues is the design of a good state-space representation scheme. Potentially many different sensor readings (average demand, response time, CPU and memory utilization, number of Java threads running, etc.) may be needed to accurately describe the system state. Historical information may also be required if there are important history-dependent effects (e.g. Java garbage collection). The state encoding should be appropriate to whatever value function approximation scheme is used, and it needs to be sufficiently compact if a lookup table is used.

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 must take correspondingly fewer updates than are acceptable in simulation. This can be addressed in part by using a compact value function representation. We also employ, as detailed below, the use of heuristics or domain knowledge to define good value function initializations; this can make it much easier and faster for RL to find the asymp-

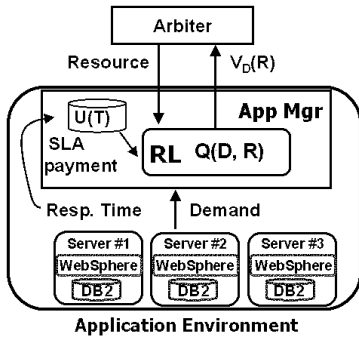


Figure 2: Operation of RL in the Application Manager.

otic optimal value function. In addition, we 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 this 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 and a poor initial policy. The latter factor can be addressed by clever initialization and hybrid training as mentioned above. In addition, one expects that some form of safeguard mechanism and/or intelligent exploration (e.g. Boltzmann or Interval Estimation) will be needed to limit penalties incurred during exploration. Perhaps surprisingly, this was unnecessary in the prototype system: a simple ϵ -greedy rule with $\epsilon = 0.1$ incurs negligible loss of expected utility. Nevertheless, intelligent/safe exploration should eventually become necessary as our system increases in complexity and realism.

RL Implementation Details

The implementation and 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 associated with a given current state of the workload and current number of servers assigned by the arbiter. 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 of 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 650.

At each time step the RL module observes the demand D , the resource level R given by the arbiter, and the SLA

payment U which is a function of average response time T . The value table is then updated using Sarsa(0), with discount parameter $\gamma = 0.5$, learning rate $\alpha = 0.2$, and using a standard decay of the learning rate for cell i by multiplying by $c/(c + v_i)$, where v_i is the number of cell visits and the constant $c = 80$. From time to time the Application Manager may be asked by the Arbiter to report its current resource value function curve $V(R)$ at the current demand, and may also receive notification that its assigned servers have changed.

It’s 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, upon RL updating of a cell’s value, any monotonicity violations with other cells in its row or column are 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 reasonable, although for certain dynamical patterns of demand variation, it may not strictly hold. Both constraints yield significantly faster and more accurate learning.

Results

This section presents typical results obtained using RL in this system. The issues of particular interest are: (1) the extent to which RL converges to stationary optimal value functions; (2) how well the system performs when using RL value estimates to do resource allocation; (3) how well the system scales to greater numbers of servers or applications.

Empirical Convergence

We now examine RL convergence by two methods. First, identical system runs are started from different initial value tables to see if RL reaches similar final states. Second, a learning run can be continued for an additional overnight training session with the cell visit counts reset to zero; this reveals whether any apparent convergence is an artifact of vanishing learning rates. While one would ideally assess convergence by measuring the Bellman error of the learned value function, this is infeasible due to the system’s highly stochastic short-term performance fluctuations (the most severe of which are due to Java garbage collection). The number of state-transition samples needed to accurately estimate Bellman error vastly exceeds the number that can be observed during overnight training.

Figure 3(a) shows RL results in an overnight run in the 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

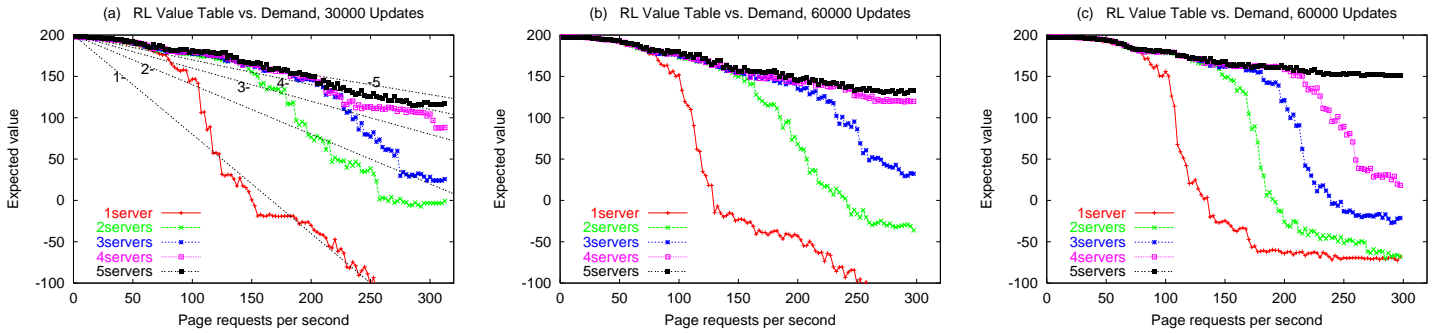


Figure 3: (a) Trade3 value function trained from heuristic initial condition, indicated by straight dashed lines. (b) Continuation from (a) with visit counts reset to zero. (c) Value function trained from uniform random initial condition.

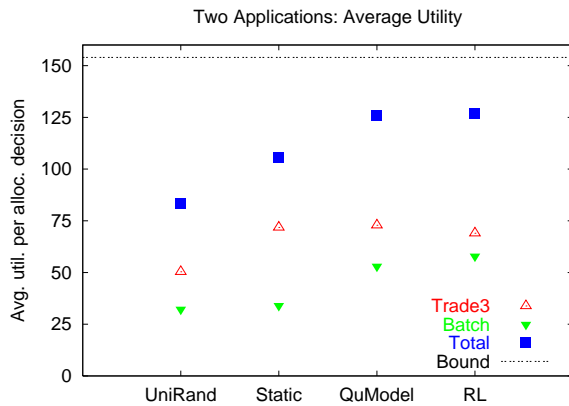


Figure 4: Performance of RL and other strategies in scenario with two applications.

performance during this entire run is high, even including the 10% random arbiter exploration.

Figure 3(b) shows results when training is continued from (a) 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.)

Figure 3(c) shows results in an equivalent amount of training time as (b), but starting from uniform random initial conditions. The solution found is for the most part comparable to (b), with differences being mainly attributable to infrequently visited non-greedy cells. This provides additional evidence of proper RL convergence, despite the radically different initial state and extremely poor initial policy in the random case.

Performance Results

Figure 4 shows the performance of RL (specifically the run shown in Figure 3(a)), in the standard two-application scenario. Performance is measured over the entire run in terms of average total system utility earned per arbiter allocation decision. This is compared in essentially identical scenarios, using identical demand generation, with three other allocation strategies. The most salient comparison is

with “QuModel,” which uses an open-loop parallel M/M/1 queuing-theoretic performance model, developed by Rajarshi Das for this prototype system with significant research effort, that estimates how changes in assigned servers would affect system performance and hence SLA payments. It makes use of standard techniques for online estimation of the model’s parameters, combined with exponential smoothing tailored specifically to address the system’s short-term performance fluctuations. Development of this model required detailed understanding of the system’s behavior, and conforms to standard practices currently employed within the systems performance modeling community.

To establish a range of possible performance values, we also plot two inferior 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.

Note that the RL performance includes all learning and exploration penalties, which are not incurred in the other approaches. It is encouraging that, in an eminently feasible amount of live training, the simple RL approach used here can obtain comparable performance to the best-practice approach for online resource allocation, while requiring significantly less system-specific knowledge to do so. Both RL and the queuing model approach are reasonably close to the maximum performance bound, which is a bit generous and most likely can be tightened to lie below the 150 level.

Scaling to Additional Applications

We have also used RL 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. This is more challenging as there are now multiple interacting RL modules, 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 training times, apparent convergence, or quality of policy compared to previous results. Performance results in this scenario are plotted in Figure 6. Once again RL performance is comparable to the queuing model approach (about 1% better, in fact), and both are quite close to the maximum possible performance.

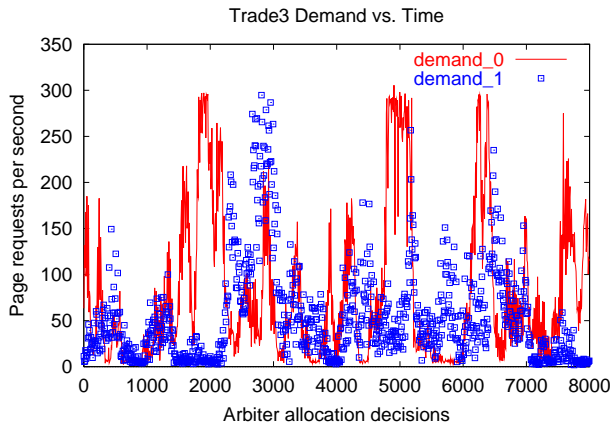


Figure 5: Two independent time-varying Trade3 demand models, used in scenario with three applications.

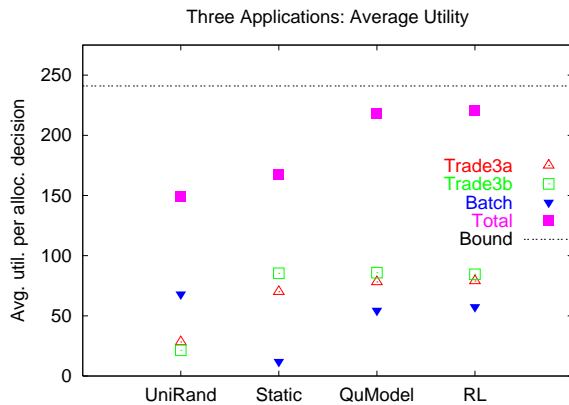


Figure 6: Performance of RL and other strategies in scenario with three applications.

While this does not establish scalability to an arbitrary number of applications, the results are encouraging and suggest that decompositional RL may well be a viable alternative to model-based approaches for real-world problems of this type. Note that global RL would be infeasible here, as the required table size would be $0.5 * 650^2 = 211\text{K}$ instead of 650, and most cells would be unvisited in an overnight run.

Conclusions

The empirical results presented in this paper provide evidence that decompositional RL may be a viable and practical method for online server allocation in real computing systems. RL can eliminate the need for explicit system performance models, while the decompositional approach may enable scaling to many applications, which would be infeasible with global Q-Learning. The evidence also plausibly suggests that such methodology may be more generally useful in other workload management scenarios, for example, when the allocated resources are storage devices or database access instead of servers.

One might have expected RL in real computing systems to require exorbitant training times, complex state descriptions and nonlinear function approximation, and to perform

poorly initially and while exploring. However, in the current prototype system, RL has turned out better than expected in all of these respects. With extremely simple state descriptions (average demand only), value function representation (a uniform grid), exploration schemes (10% random allocations), and initialization schemes (linear in demand per server), RL achieves performance comparable to our best efforts within the model-based framework, and does so in a quite feasible amount of training time.

Our current work examines effects arising when there are significant delays in reallocating servers — these are negligible in the current system. As delayed rewards are central to RL’s basic formalism, it should continue to perform well in this case. We also continue to study scalability of applications, servers, and state descriptions, and plan to address this in part using nonlinear value function approximators. Hybrid training and intelligent exploration will also be used to obtain high initial performance and safe action exploration.

There is also much interesting analysis to be done in establishing conditions under which decompositional RL converges to correct value functions, and if not, what is the minimal state-space augmentation that would enable it to do so. One case where purely local Sarsa may be provably convergent is if the other applications provide stationary additive noise to the value function optimized by the global policy.

References

- Dietterich, T. G. 2000. Hierarchical reinforcement learning with the MAXQ value function decomposition. *J. of AI Research* 13:227–303.
- Guestrin, C.; Lagoudakis, M.; and Parr, R. 2002. Coordinated reinforcement learning. In *Proceedings of ICML-02*, 227–234.
- ICAC. 2004. *Proceedings of the First International Conference on Autonomic Computing*. IEEE Computer Society.
- Kearns, M. J., and Koller, D. 1999. Efficient reinforcement learning in factored MDPs. In *Proceedings of IJCAI-99*, 740–747.
- Kephart, J. O., and Chess, D. M. 2003. The vision of autonomic computing. *Computer* 36(1):41–52.
- Meuleau, N.; Hauskrecht, M.; Kim, K.-E.; Peshkin, L.; Kaelbling, L.; Dean, T.; and Boutilier, C. 1998. Solving very large weakly coupled Markov Decision Processes. In *Proceedings of AAAI-98*, 165–172.
- Russell, S., and Zimdars, A. L. 2003. Q-decomposition for reinforcement learning agents. In *Proceedings of ICML-03*, 656–663.
- Singh, S., and Cohn, D. 1998. How to dynamically merge Markov Decision Processes. In Jordan, M. I.; Kearns, M. J.; and Solla, S. A., eds., *Advances in Neural Information Processing Systems*, volume 10, 1057–1063. MIT Press.
- 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.
- Trade3. 2004. Websphere benchmark sample. <http://www.ibm.com/software/webservers/appserv/benchmark3.html>.
- Walsh, W. E.; Tesauro, G.; Kephart, J. O.; and Das, R. 2004. Utility functions in autonomic systems. In *1st IEEE International Conference on Autonomic Computing*, 70–77.