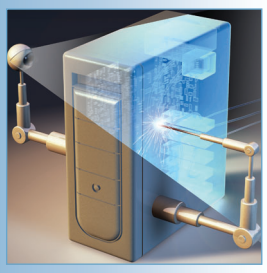

Reinforcement Learning in Autonomic Computing

A Manifesto and Case Studies

Reinforcement learning (RL) is a promising new approach for automatically developing effective policies for real-time self-* management. RL has the potential to achieve superior performance to traditional methods while requiring less built-in domain knowledge. Several case studies from real and simulated systems-management applications demonstrate RL's promises and challenges. These studies show that standard online RL can learn effective policies in feasible training times. Moreover, a Hybrid RL approach can profit from any knowledge contained in an existing policy by training on the policy's observable behavior without needing to interface directly to such knowledge.

Gerald Tesauro
IBM T.J. Watson Research Center

Much current work on self-managing systems aims to engineer human expert domain knowledge into various kinds of automated self-* management policies. For example, in real-time system-performance management, much research effort focuses on designing and constructing explicit system-performance models, such as queuing-theoretic[1] or control-theoretic[2] models, that effectively optimize in real time the performance of complex computing systems.

Domain knowledge is a prominent feature of Figure 1, which depicts an IBM-proposed vision[3] of autonomic computing's general operation. The figure envisions a generic relationship between an *autonomic manager* software element and a (hardware or software) *managed element*. The autonomic manager continually executes a *monitor-analyze-plan-execute* (MAPE) loop in which it observes sensor readings, analyzes and plans an appropriate management decision, and then executes that decision via effectors. A central knowledge base, accessible by the other components, contains knowledge pertaining to the likely effectiveness of various possible management decisions in achieving the manager's overall policy objectives. Typically, such knowledge takes the form of an explicit system model, which estimates how various management decisions would affect sub-

 1089-7801/07/$25.00

sequent states of the managed element, and might also include forecasts of any external processes that generate workflow or traffic processed by the managed element.

A key factor limiting rapid adoption and wide usage of self-* managing systems such as the system in Figure 1 is the difficulty of engineering a sufficiently accurate knowledge module that can achieve acceptable performance in deployed systems. Because today's computing systems are highly complex and distributed, developing accurate models of them is a potentially complex and time-consuming task. Moreover, developing such models might require original research. For example, queuing network models of multitier Internet services have only recently appeared in the literature.[4]

In particular, researchers are only beginning to approach proper treatment of the full range of computing systems' complex dynamic behaviors within standard model-building frameworks. Standard control models, for example, model such effects only approximately, and standard queuing models ignore them entirely given that they assume steady-state system behavior.

A further challenge for model-based approaches is that today's computing systems, as well as the workflow characteristics and business processes that they support, are continually evolving, so periodic redesign of the knowledge modules of self-* systems will be necessary.

Defeating the Knowledge Bottleneck with Machine Learning

Machine learning (ML) might hold great promise in overcoming the knowledge bottleneck just described. ML is a subfield of artificial intelligence that aims to develop methods for automatically acquiring knowledge from data. For example, the broad paradigm of *supervised learning* uses a data set of (input, output) pairs to learn a classification or regression model exhibiting a deeper "understanding" of the relation between inputs and outputs beyond the specific training exemplars. Ideally, the learning algorithm will correctly generalize to novel exemplars not seen during training. Likewise, *unsupervised learning* is another broad paradigm that uses input exemplars only (no target outputs), with the goal of discovering previously unknown structures or relationships between exemplars or their components. Such knowledge can be useful in data mining or for grouping exemplars into closely related clusters.

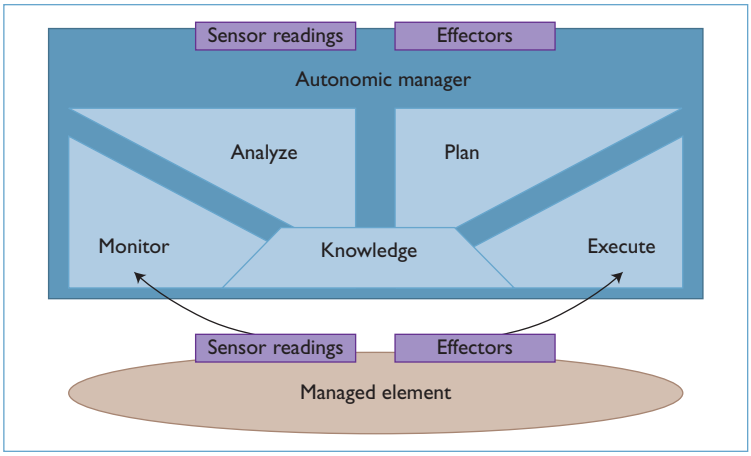

Figure 1. A standard autonomic computing monitor-analyze-plan-execute (MAPE) loop. The autonomic manager continually monitors sensor readings, analyzes and plans management decisions, and executes the decisions via effectors. The MAPE components have access to a central knowledge base containing information pertaining to the likely effectiveness of different management decisions in achieving policy objectives.

To overcome the knowledge bottleneck in developing self-* systems, ML approaches would ideally use so-called tabula rasa learning methods — that is, methods that learn with little or no built-in domain knowledge. Having to build significant knowledge into the learning algorithm's architecture would defeat the purpose. However, given that some level of domain knowledge is usually available for most systems, and that completely knowledge-free learning might be impractical for various reasons, another desideratum of machine-learning methods is that they can also gracefully incorporate any available initial domain knowledge.

Consider the ML paradigms that would be appropriate for learning self-* management policies — that is, mappings from system states to selected management actions. (Note that this definition of "policy" differs from other more general meanings used in autonomic computing.) *Adaptive control*,[5] one such paradigm already used in systems management, fits readily into the model-based control theory framework. Adaptive control methods aim to automatically and continuously adapt model parameters as system characteristics change, and to automatically perform system identification — that is, develop a model of how control actions influence the system state's evolution. To the extent that such efforts require little engineering of domain-specific knowledge, they can contribute to our overall goal of avoiding the knowledge bottleneck.

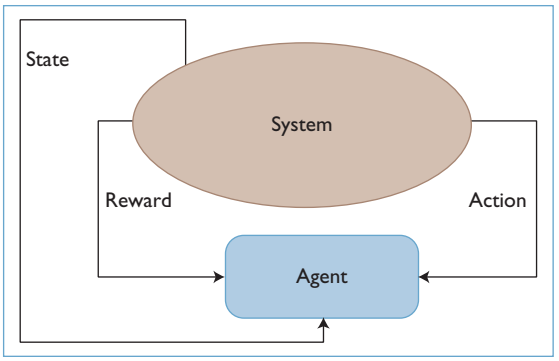

Figure 2. A standard reinforcement learning (RL) interaction loop. An agent learns effective decision-making policies through trial-and-error interactions with its environment.

However, I advocate a more ambitious ML paradigm, *reinforcement learning* (RL), which is largely unstudied for systems-management applications. In its most commonly studied form, RL aims to learn effective management policies in the absence of explicit system models, with little or no domain-specific initial knowledge. This type of approach attempts to circumvent knowledge bottlenecks and thus provides a primary focus for this article. Other variants of RL might also be of interest for autonomic computing, but they're beyond this article's scope. *Model-based reinforcement learning* approaches, such as Sutton's Dyna-Q algorithm,[6] aim to simultaneously learn system models and management policies. Such approaches relate more closely to adaptive control methods, and perhaps the most significant distinction lies in the difference between typical control-theoretic objective functions and the RL objective of maximizing cumulative future reward.

A final caveat is worth reiterating regarding the potential impracticality of pure knowledge-free learning approaches. Such approaches could entail running arbitrarily poor initial policies in the live system. These considerations in my own research have in fact led to an alternative hybrid RL approach (described later), which exploits knowledge contained in an external policy without needing to directly interface with such knowledge.

Reinforcement Learning: Promises and Challenges

RL provides a novel approach to systems management that differs radically from more traditional explicit model-building approaches. RL ideally offers a way to avoid knowledge bottlenecks by automatically learning high-quality management policies with little or no built-in system-specific knowledge. Moreover, RL is grounded in a powerful sequential decision theory; in principle, it should be able to learn optimal policies, fully accounting for a decision's long-range dynamic consequences.

In RL's normal operation, illustrated in Figure 2, an agent learns effective decision-making policies through an online trial-and-error process in which it interacts with an environment. Assuming time progresses in discrete steps, each interaction consists of

- observing the environment's current state s_t at time t,
- performing some legal action a_t in state s_t, and
- receiving a reward r_t (a numerical value that the user would like to maximize) followed by an observed transition to a new state s_{t+1}.

The Sarsa[6] rule is an example RL algorithm that learns a value function $Q(s, a)$ estimating an agent's long-range expected value, starting in state s and taking initial action a. Sarsa operates in a step-by-step fashion: at each time step t, after having observed a transition from the previous state s_{t-1} to the current state s_t, Sarsa incrementally adjusts the estimated value of the prior state-action pair, $Q(s_{t-1}, a_{t-1})$, as follows:

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

Here (s_{t-1}, a_{t-1}) are the initial state and action at time $t - 1$; r_{t-1} is the immediate reward at time $t - 1$; (s_t, a_t) denotes the next state and next action at time t; 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. Several interesting and powerful theorems guarantee that algorithms such as Sarsa learn optimal value functions and optimal policies for Markov decision process (MDP) environments when lookup tables are used to represent the value function. We've also seen notably successful applications of RL over the past decade in real-world problems ranging from helicopter control to financial-markets trading to world-championship game playing.[7-9]

General Issues for Successful RL Applications

Achieving practical success with RL in autonomic computing systems requires that we address four commonly identified issues:

- RL might need to observe a huge number of (state, action) pairs and state transitions to converge to optimal policies. The sample complexity depends in part on how noisy the immediate rewards and state transitions are and, to a much greater degree, on the compactness of the value function representation.
- When training online in a live system, any poor decisions RL makes before it has learned a good policy can result in quite poor rewards, and the cost of this can prohibit an online training approach.
- RL methods generally need to include a certain amount of *exploration* of actions believed to be suboptimal, purely to facilitate better learning of value functions and policies. As with poor initial policies, exploratory actions can be costly in live systems, so the user must take care that such costs aren't prohibitive.
- Real-world problems might not be strict MDPs — they can exhibit incomplete observability, history dependence, and nonstationarity — so researchers might need to modify standard RL to deal with such non-Markovian aspects.

In surveying major application success stories, several common clues or themes emerge regarding techniques for successfully addressing these challenges. One of the most important themes is how to represent the learned value function or policy. Lookup table representations are generally inadequate because their size scales exponentially with the state space dimensionality. They also provide no way to learn about a particular (state, action) pair except by actually visiting the state and trying the given action, which is infeasible in large state spaces. Hence, some form of compact value function representation is necessary.

Using knowledge of a given problem domain, we could tailor a specific structured representation (such as a particular polynomial form) appropriate to the problem, and then RL need only learn coefficients within this structure. However, for tabula rasa learning, the method of choice is to use a flexible nonparametric regression scheme capable of approximating a wide variety of function types. Many such schemes are available — for example, splines, neural networks, and regression trees. Although combinations of RL and function approximators generally lack convergence guarantees, they often work well in practice. Some guarantees exist for learning while using a stationary policy — for example, a *residual gradient* method[10] guarantees convergence to a local minimum in approximation error.

The use of function approximation obviates visiting every state in the state space. Upon visiting a tiny fraction of the full state space (scaling with the complexity of the particular approximator), the function approximator might then generalize well enough in other parts of the state space that it didn't see during training. Likewise, function approximation can provide a mechanism for generalizing across actions, greatly reducing the need for exploratory actions relative to the lookup table case. This particularly holds when using compact action representations that encourage generalization. In resource-allocation tasks, for example, we can represent resource levels using a single scalar input. By training on certain resource levels, the function approximator can make inferences about the expected value at other resource levels.

Another important lesson from application successes is that training in simulation, or offline on a fixed data set, is preferable to online training in a live system. This is due to several factors. First, when training in simulation, poor decisions due to a poor policy or exploration invoke only simulated costs, whereas such costs are real and can be prohibitive in a live system. Second, learning in a simulator can be made more effective by placing the system in certain states that are either difficult to reach or excessively dangerous to explore in the live system. Third, the physical time scales associated with real policy decisions and state changes can make online learning unacceptably slow in terms of elapsed time, whereas in simulation or offline training, time steps might be orders of magnitude faster than physical time scales.

Applying RL to Autonomic Computing

Let's now consider prospects for applying RL within autonomic computing. Comparing the RL loop in Figure 2 with the MAPE loop in Figure 1, the operation of RL within MAPE appears direct and natural, assuming the sensors provide relevant state descriptions and reward signals. In fact, if the managed element comprises an MDP environment, RL should be able to learn an optimal management policy. The chief distinction might be that RL policies are generally characterized in the planning community as *reactive* planners — that is, they make decisions fairly immediately without explicit search or forecasting of future states. MAPE's

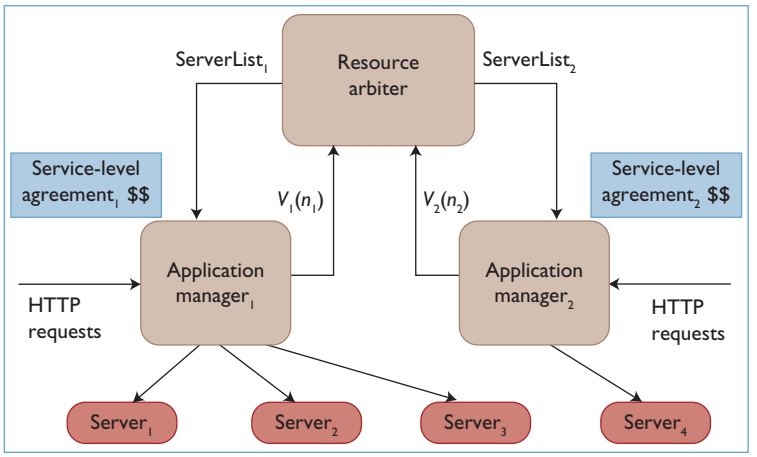

Figure 3. Resource allocation scenario in prototype data center. Each application manager module periodically communicates expected utility information to a resource arbiter module, which then computes an optimal allocation of servers to applications.

plan component, however, allows more general deliberative and generative planning procedures. Hence, RL might not provide an effective solution for problems in autonomic computing that require generative planning. However, RL's reactive planning is quite sophisticated and might be perfectly adequate for many problems in which generative planners are currently used or envisioned. In any case, reactive planning is likely to be essential for real-time systems-management applications.

RL in Systems Management: Case Studies

Several recent case studies demonstrate both promise and interesting challenges in applying various RL approaches to real-time systems-management problems. The first two studies described here summarize my own work aimed at developing practical, scalable RL methodologies for real-time performance management.

As part of our laboratory's research, my IBM colleagues David Chess, Ian Whalley, Rajarshi Das, and I developed a prototype data-center testbed (see Figure 3) for studying various approaches to dynamically allocating servers among multiple applications. The testbed uses real HTTP servers, realistic Web-based workloads (primarily Trade3, a realistic emulation of online stock trading), and a stochastic time-series model derived from real Web traffic[11] to generate time-varying demand in each application. Our demand generator supports both open-loop Poisson traffic, with a variable mean arrival rate, and closed-loop traffic, in which we simulate a finite population of customers with a fixed think-time distribution, and can dynamically vary the number of customers. Das developed queuing models for each of these modes using standard best-practice modeling and estimation techniques. In brief, these models use current measurements such as mean arrival rate λ_t and mean response time τ_t, combined with exponentially smoothed parameter estimates such as service rate μ_t, to estimate new asymptotic performance levels under various hypothetical changes in the number of allocated servers, assuming steady-state dynamics. Additional details of these models appear elsewhere.[12] Such models provide a solid benchmark strategy, as well as an initial policy for the Hybrid RL approach described later.

In this testbed, each application has its own *application manager* module, which communicates with a *resource arbiter* module regarding resource needs. Allocation decisions are made in fixed 5-second time intervals, in which each application manager i reports to the arbiter a commonly scaled utility curve $\{V_i(n_i), n_{min} \le n_i \le n_{max}\}$ that estimates expected business value, in the application's current state, as a function of the number n_i of allocated servers. We assume that an application's business value is defined in monetary units by expected revenue according to the application's service-level agreement (SLA), which stipulates payments or penalties as a function of one or more performance metrics. The application managers periodically recompute utility curves as the applications' states and load levels fluctuate. Upon receiving the current utility curves, the arbiter solves for the globally optimal allocation maximizing total expected value. It then conveys a list of assigned servers to each application, which the application uses in a dedicated fashion until the next allocation decision.

Online RL in the Data Center Testbed

In my initial research,[13] I devised a decompositional version of standard online RL, with purely local learning within each Trade3 application and no learning at the global decision-making level. This achieves scalability to many applications, but unfortunately isn't covered by existing convergence proofs, so convergence of the approach needs empirical study. To make a tabular value function feasible, I made a severe approximation in representing the application state s_t solely by the (discretized) current mean arrival rate λ_t of page requests, ignoring several other sensor read-

ings (mean response times, queue lengths, and so on) that provide potentially useful state information. (We estimate λ_t as the number of requests that arrived in the most recent interval, divided by the length of the interval [5 seconds].) Because the action is simply the number of allocated servers n_t, this enables a feasible 2D grid representation of the value function $Q = Q(\lambda_t, n_t)$.

Good initialization is vital for online RL, and for this purpose, I devised a heuristic initialization in which Q depends linearly on demand per server λ/n. Such initialization required a modicum of elementary domain knowledge. I also implemented a standard ε-greedy exploration rule, in which the arbiter chooses a random allocation with probability $\varepsilon = 0.1$ instead of the utility-maximizing allocation. These standard tricks of the trade in RL worked well in our small testbed, but seem less feasible in more complex larger-scale systems.

Figure 4 shows performance results comparing online RL in overnight training runs to three other strategies for two applications (one Trade3 and one non-Web-based batch application, with steady resource valuation) and three applications (two Trade3s and one batch). The three alternative strategies are

- *UniRand*, which makes uniform random arbiter allocations,
- *Static*, which denotes the best static allocation, and
- *QuModel*, which uses Das's open queuing network model.

The dashed line indicates an analytical upper bound on the best possible performance that this system can achieve using the observable information.

The RL performance includes all learning and exploration penalties, which the other approaches don't incur. It's 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.

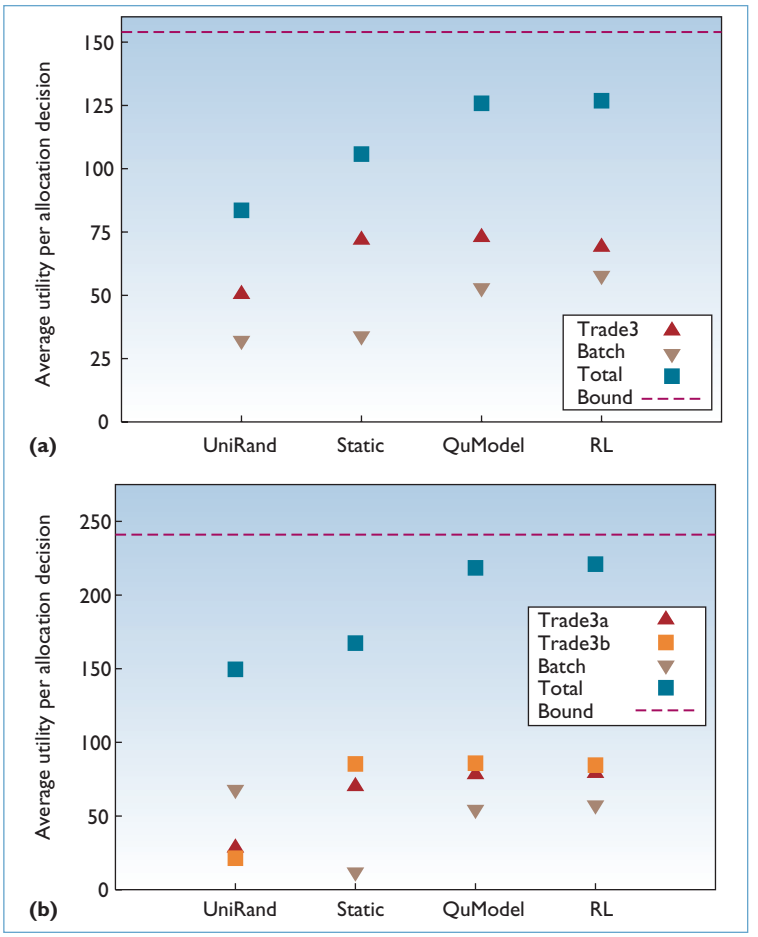

Figure 4. Online RL performance results. (a) Average utility of RL and other strategies in a scenario with two applications. (b) Average utility in a scenario with three applications.

Hybrid RL in Data Center Testbed

Motivated by potential scalability problems with the online RL approach, I've worked more recently with University of Texas graduate student Nicholas Jong to devise a more practical and intriguing Hybrid RL approach. Hybrid RL can piggyback on top of any existing knowledge or heuristics built into an external policy, but doesn't need to directly interface to such knowledge. Instead, Hybrid RL needs only to observe the policy's behavior.

In the Hybrid RL method, we first run an overnight experiment using some initial policy (for example, a policy based on queuing models in each Trade3 application). We record all (state, action, reward) tuples observed in each application and use this data to do offline batch RL training of local value functions for each Trade3. Theory suggests that if RL works properly on these data sets, the resulting RL-based policy will outperform any imperfect initial policy.

This approach completely avoids poor performance (and concomitant need of clever initialization) that could occur with live online training. The other key aspect of Hybrid RL is that we use nonlinear function approximators in place of

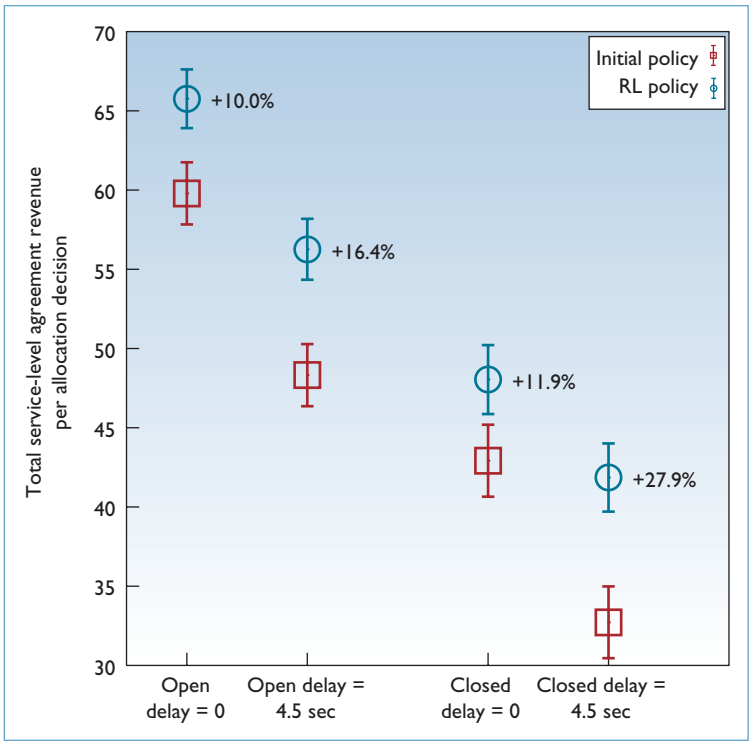

Figure 5. Hybrid RL performance results. The graph shows a comparison of delay = 4.5 seconds with delay = 0 results in open-loop and closed-loop traffic scenarios.

lookup tables. As I described earlier, function approximators provide a mechanism for generalizing training experience across states. They also generalize across actions, greatly reducing the need for exploratory actions. In fact, with our system, we obtain improved policies without any exploration by training solely on the model-based policy decisions.

In our system, Hybrid RL consistently achieves substantial performance improvements over a wide variety of open and closed queuing network models, and its performance advantage widens when large switching delays in reassigning servers are present. To handle switching delays, we included an extra lagged variable, the previous server allocation n_{t-1}, as part of the state representation at time t. This enables learning different values for receiving n servers depending on how many the learner currently has — in effect, encoding the cost of changing the number of servers in differences between such value estimates.

Figure 5 shows an example of results using Hybrid RL. The figure compares our zero-delay results, using our best open and closed queuing network models, with corresponding experiments that impose a delay (downtime) of 4.5 seconds when a server is reassigned to a different application. We chose the delay to be a huge fraction of the 5-second allocation interval so that its empirical effects would be as clear as possible. In the zero-delay experiments, each Hybrid RL policy achieved double-digit percentage improvement in total utility over its queuing-model counterpart (10.0 percent in the open-loop traffic experiment and 11.9 percent in the closed-loop experiment). Adding delay substantially harms the average performance in all cases. However, the amount of policy improvement Hybrid RL exhibits over its initial policy increases in both absolute and relative terms. In the open-loop scenario, the improvement increases to 16.4 percent, whereas in the closed-loop traffic scenario, the improvement jumps to 27.9 percent.

Apart from raw performance improvements, we've obtained several interesting insights regarding how Hybrid RL can outperform the initial queuing-model policies. (For complete details, please see our recent ICAC-2006 paper.[12]) One insight is that, for reasons too technical to detail here, the queuing model policies have a bias toward overprovisioning due to interactions of their response-time estimates with the nonlinear SLA function. However, by learning to estimate utility directly, the RL nets can achieve less biased estimation errors.

The second insight is that, whereas steady-state queuing models can't properly treat transients and switching delays, our RL nets can. The learned policies exhibit hysteresis — that is, a tendency to prefer steady allocations over switching based on instantaneous state.

Third, RL policies exhibit greatly reduced thrashing, which can occur using queuing models when the system is overloaded and the models overestimate how many servers they need to obtain good performance. Reduction in thrashing appears to result partly from the RL policies not overestimating resource needs, partly because of high estimated switching costs for large changes in the number of servers and partly by directly training on the thrashing episodes generated by the queuing models and directly learning that thrashing behavior yields poor utility.

Additional Case Studies

Other recent work outside of IBM also shows early promise in applying RL paradigms to systems-performance management. David Vengerov and Nikolai Iakovlev of Sun Microsystems applied online RL combined with function approximation

to learn policies for real-time allocation of processor cores among two partitions on a Sunfire 2900 machine running Solaris 10, based on stationary random fluctuations in the number of threads running in each partition.[14] Their function approximator used a *fuzzy rulebase*, an intriguing approach that employs weighted combinations of basis functions capable of highly nonlocal inference. Their experiments in the live system indicated that their RL policies achieved near-optimal performance in two different scenarios, one with zero switching cost and one with high switching cost. By contrast, two fixed-baseline strategies respectively excelled in one scenario but did poorly in the other.

Shimon Whiteson and Peter Stone of the University of Texas used RL in a simulated multitier computing system comprising multiple load balancers, Web servers, mail servers, and databases.[15] The task was to route user requests to machines and schedule execution on machines so as to maximize importance-weighted performance (different users have different importance levels). They devised an RL method to automatically learn a routing policy for this system, and then, using the router's learned value function, they devised a heuristic insertion scheduler that inserts arriving jobs at the location in the queue with the highest estimated value. They compared both the learned router and scheduler against several reasonable fixed-heuristic strategies. In several experiments involving progressively more complex networks, they showed that the learned routing and scheduling policies consistently obtained significant improvement over the heuristic policies, either individually or in tandem.

Vengerov also used RL in a simulator to learn utility-based schedulers in soft real-time systems, in which the utility of completing a job decays with time.[16] The approach again combined RL with a fuzzy rulebase function approximator. He used the approach on single and multiple machines for both preemptive scheduling and oversubscribing — that is, squeezing a job onto a machine with fewer free CPUs than the job desires. For both preemption and oversubscribing, the approach computed alternate possible machine states resulting from various possible preemption or oversubscribing actions and selected the action leading to highest expected value. Simulations showed that the learned policies obtained a slight advantage over the best heuristic aggressive oversubscribing/preemptive policy in lightly loaded conditions, and that the advantage of learning widened when the system was more heavily stressed.

Within the broad realm of real-time self-* system-management problems, RL is a novel and radically different approach that holds great promise for wide practical use. RL's power derives from two key ingredients:

- Because of its grounding in the MDP framework, RL can, in principle, obtain truly decision-theoretic optimal policies that properly account for a decision's future effects.
- RL can solve MDPs without an explicit model of the MDP, relying instead on trial-and-error observations of interactions with the environment.

Based on these two ingredients, RL promises to outperform other less-principled decision-making methods and require less system-specific knowledge.

Prior to the case studies I've described, we might have expected and feared that real computing systems would prove too challenging for RL: training times could be exorbitant, state descriptions might be intractably complex, and suboptimal performance while learning and exploring could be prohibitively costly. Although all these factors might still arise, the early results from the case studies are promising and suggest that these factors are perhaps not as bad as expected. The case studies all found that simple approximate state descriptions worked well in practice, and that required training times were generally feasible. Also, one case study found online learning and exploration costs to be tolerable given a simple heuristic value function initialization.[13]

The Hybrid RL approach, in particular, takes a great stride toward practical use of RL by marrying it in an elegant way with best-practice knowledge-engineering techniques or with whatever knowledge happens to be available in existing systems-management policies. Importantly, Hybrid RL doesn't need to directly interface to such knowledge or have it built into its internal architecture. Instead, Hybrid RL need only observe the external policy's behavioral consequences in terms of rewards and state transitions. As a result, Hybrid RL could work with virtually any initial policy or controller, ranging from a random policy to a crude heuristic to a highly sophisticated multitier

queuing network model to even a human decision maker's actions. Hybrid RL also provides an interesting way to tackle nonstationarity in problem domains, which other studies don't address. When a system undergoes a major change, such as hardware upgrades or changes in the SLA, value-function retraining is often necessary. In this case, Hybrid RL can fall back on the model-based policy to deliver acceptable performance while accumulating a new training set for retraining.

The time now seems ripe for a serious attempt to use RL in a deployed or deployable system. A good way to approach this would be to extract training data from log files generated by a deployed system, run batch RL on the training data, and then compare the policy decisions recommended by RL with the actual policy decisions made in the log data. Human domain experts might be able to assess whether such policy differences would likely improve or harm the system's performance before actually deploying the learned policy.

It's also worth examining RL in broader types of systems-management problems. The most promising applications would have a tractable state-space representation, frequent online decision making depending on time-varying system state, frequent observation of numerical rewards in an immediate or moderately delayed relation to management actions, and preexisting policies that obtain acceptable performance. Many performance-management applications clearly have such properties. Among them are dynamic allocation of other resource types, such as bandwidth, memory, threads, and logical partitions. Other candidates include online performance tuning of system control parameters, such as Web server, operating system, and database parameters. Finally, RL could encompass simultaneous management to multiple criteria (for example, performance and availability), as long as the rewards pertaining to each criterion were on an equivalent numerical scale.

References

1. D.A, Menascé, V.A.F. Almedia, and L.W. Dowdy, *Performance by Design: Computer Capacity Planning by Example*, Prentice Hall, 2004.
2. J.L. Hellerstein et al., *Feedback Control of Computing Systems*, Wiley & Sons, 2004.
3. J.O. Kephart and D.M. Chess, "The Vision of Autonomic Computing," *Computer*, vol. 36, no. 1, 2003, pp. 41–52.
4. B. Urgaonkar et al., "An Analytical Model for Multi-Tier Internet Services and Its Applications," *Proc. Sigmetrics*, ACM Press, 2005, pp. 291–302.
5. K.J. Astrom and B. Wittenmark, *Adaptive Control*, 2nd ed., Prentice Hall, 1994.
6. R.S. Sutton and A.G. Barto, *Reinforcement Learning: An Introduction*, MIT Press, 1998.
7. G. Tesauro, "Temporal Difference Learning and TD-Gammon," *Comm. ACM*, vol. 38, no. 3, 1995, pp. 58–68.
8. J. Moody and M. Saffell, "Learning to Trade via Direct Reinforcement," *IEEE Trans. Neural Networks*, vol. 12, no. 4, 2001, pp. 875–889.
9. A.Y. Ng et al., "Inverted Autonomous Helicopter Flight via Reinforcement Learning," *Proc. Int'l Symp. Experimental Robotics*, 2004; http://ai.stanford.edu/~ang/papers/iser04-invertedflight.pdf.
10. L. Baird, "Residual Algorithms: Reinforcement Learning with Function Approximation," *Proc. Int'l Conf. Machine Learning* (ICML), ACM Press, 1995, pp. 30–37.
11. M.S. Squillante, D.D. Yao, and L. Zhang, "Internet Traffic: Periodicity, Tail Behavior, and Performance Implications," *System Performance Evaluation: Methodologies and Applications*, E. Gelenbe, ed., CRC Press, 1999, pp. 23–37.
12. G. Tesauro et al., "A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation," *Proc. Int'l Conf. Autonomic Computing* (ICAC), IEEE CS Press, 2006, pp. 65–73.
13. G. Tesauro, "Online Resource Allocation Using Decompositional Reinforcement Learning," *Proc. AAAI*, 2005, pp. 886–891.
14. D. Vengerov and N. Iakovlev, "A Reinforcement Learning Framework for Dynamic Resource Allocation: First Results," *Proc. Int'l Conf. Autonomic Computing* (ICAC), IEEE CS Press, 2005, pp. 339–340.
15. S. Whiteson and P. Stone, "Adaptive Job Routing and Scheduling," *Eng. Applications of Artificial Intelligence*, vol. 17, no. 7, 2004, pp. 855–869.
16. D. Vengerov, *A Reinforcement Learning Framework for Utility-based Scheduling in Resource-Constrained Systems*, tech. report TR-2005-141, Sun Microsystems, 2005.

Gerald Tesauro is a research staff member at the IBM T.J. Watson Research Center. His research interests have included theoretical and applied machine learning in a wide variety of settings, including multiagent learning, dimensionality reduction, credit scoring, computer virus recognition, computer chess (Deep Blue), intelligent e-commerce agents, and TD-Gammon, a self-teaching program that learned to play backgammon at human world-championship level. He is currently interested in exploring potential wide applicability of ML approaches throughout the huge emerging domain of self-managing computing systems. Tesauro has a PhD in theoretical physics from Princeton University. Contact him at gtesauro@us.ibm.com.