A Reinforcement Learning Approach to Dynamic Resource Allocation*

David Vengerov
Sun Microsystems Laboratories
UMPK16-160
16 Network Circle
Menlo Park, CA 94025
david.vengerov@sun.com

Abstract

This paper presents a general framework for performing adaptive reconfiguration of a distributed system based on maximizing the long-term business value, defined as the discounted sum of all future rewards and penalties. The problem of dynamic resource allocation among multiple entities sharing a common set of resources is used as an example. A specific architecture (DRA-FRL) is presented, which uses the emerging methodology of reinforcement learning in conjunction with fuzzy rulebases to achieve the desired objective. This architecture can work in the context of existing resource allocation policies and learn the values of the states that the system encounters under these policies. Once the learning process begins to converge, the user can allow the DRA-FRL architecture to make some additional resource allocation decisions or override the ones suggested by the existing policies so as to improve the long-term business value of the system. The DRA-FRL architecture can also be deployed in an environment without any existing resource allocation policies. An implementation of the DRA-FRL architecture in Solaris 10 demonstrated a robust performance improvement in the problem of dynamically migrating CPUs and memory blocks between three resource partitions so as to match the stochastically changing

*This material is based upon work supported by DARPA under Contract No. NBCH3039002.

workload in each partition, both in the presence and in the absence of resource migration costs.

Keywords: Reinforcement Learning, Utility Computing, Resource Allocation, Fuzzy Logic

1. Introduction

Developing computing systems that are self-configuring and self-optimizing in the face of unpredictable environmental stress conditions is becoming the central concern of industrial giants such as Microsoft, IBM, HP, and Sun. For example, the recently popular concept of "utility computing" refers to one of the capabilities required to achieve the above vision in a large-scale distributed computing system. The utility computing paradigm describes the environment where system components (agents) "trade" resources with each other so as to respond better to changes in external demand.

While this vision is very promising, many of its components still need to be adequately developed. Developing self-adaptive policies for dynamic resource allocation is the key issue in developing self-optimizing utility computing systems. Sun Microsystems has enabled this capability in its Solaris (TM) 10 Operating System [6] and the DRA-FRL architecture presented in this paper is evaluated in that domain. Other experimental studies can be found in papers from IBM [12] and HP [3], which serve as representative examples of experiments conducted in this area by other industry researchers. Both of these papers agree that the central issue that needs to be resolved for successful dynamic resource allocation among multiple competing entities is that of predicting the future utility that each of them is going to obtain for a given allocation. This is generally true for any type of reconfiguration or adaptation in a system, not just for making resource allocation decisions. Both papers suggest that the long-term business value (system rewards due to service level agreements) should be the ultimate objective, but none of them adequately solve this problem.

Reinforcement learning (RL) is the emerging solution approach for making decisions based on statistically estimating and maximizing the expected *long-term* utilities [2,5,7]. The principal contribution of this paper is a demonstration of how reinforcement learning can be used for learning utility functions for making dynamic resource allocation (or any system reconfiguration) decisions in unknown stochastic

dynamic environments with very large or continuous state spaces.

2 Problem Formulation

Consider a computing facility with a pool of shared resources (CPUs, memory, bandwidth, storage space, mobile servers, etc.). Such a facility can be a service grid, a data center, a supercomputer, or a multi-processor machine running an operating system supporting dynamic reconfiguration of resources (e.g., Solaris 10). Assume that N clients (projects) are using this facility simultaneously and that resources can be dynamically reassigned among the projects, but only one project at a time can use a given resource.

Consider a simplified scenario when resources can be migrated instantaneously and at no cost. Then, it would suffice to migrate resources between projects in response to current (or recently observed) conditions only if migration decisions can be initiated instantaneously as soon as any load imbalance is observed. However, if the re-evaluations of system conditions are infrequent enough that a non-negligible amount of work can be done between the re-evaluation points, then a forward-looking resource reassignment policy is required. That is, a chunk of resources should be migrated from project i to project j only if the expected utility gain of project j during the *next time interval* outweighs the expected utility loss of project i during that interval.

Now consider a more realistic scenario when resource migrations require a non-negligible time during which resources are idling (or are not fully utilized) or if there is some cost associated with resource migrations. In this case, a chunk of resources should be migrated from project i to project j only if the expected *long-term* utility gain of project j outweighs the expected *long-term* utility loss of project i. That is, a poor resource allocation decision might require another reassignment at the very next decision point, incurring another reassignment cost. Also, the backlog of waiting jobs can significantly increase during the time interval when a poor resource allocation decision is made, and it might take many time steps to reduce this backlog. Therefore, resource allocation decisions should consider not only the immediate benefit they bring to the system during the next time interval but also the long-term effects in

terms of future migration costs and demand-resource match.

The goal is to develop a scalable algorithm for reassigning multiple resource units between projects so as to maximize the productivity of the computing facility (average utility per time step received from all the shared resources) under *unknown* job arrival rates for different projects and *unknown* probability distribution of job characteristics (job length, job size, etc.).

3 Solution Methodology

3.1 Overview

Let $U^i(s)$ for project i be the expected future utility per unit of time received by that project starting from the state s. Once such utility functions are obtained, a large variety of resource allocation methods can be used to reallocate resources among the projects so as to maximize some function of total project utilities such as $\sum_i U^i - C$, where C is the total future resource transfer cost.

The key problem addressed in this paper is that of learning $U^i(s)$ in the dynamic resource allocation setting. If project resource allocations are fixed, then each project is faced with a stochastic policy evaluation problem which can be solved using the Reinforcement Learning (RL) methodology [7]. However, if resources allocated to one project depend on what is happening with other projects, then long-range dependencies arise between project states and the problem faced by each project loses its Markov property. Even though RL convergence proofs have only been developed for Markovian domains, some researchers (e.g., [8, 11]) have recently demonstrated experimental results suggesting that RL can still be used to learn good dynamic resource allocation policies in non-Markovian domains.

The work in [8] considers a simple scenario where the state of each project is described by a single variable that can take only a small number of values and then uses a table-based RL algorithm suitable for this simple scenario. The work in [11] considers a more complex problem where the state of each project is described by two real-valued variables, which requires a combination of standard reinforcement learning with utility function approximation architectures for generalizing utility values to states that have never been encountered before. Both of the above papers consider transfers of only a single

resource. The current paper extends the work in [11] by considering transfers of multiple resource types. While the experimental results are presented only for the case of two resource types, one can easily see that our approach directly applies to the case of multiple resource types.

Dynamic RL-based allocation of multiple resource types has not been considered before because the size of the state space increases exponentially with the number of resource types, making it very difficult for RL to learn good utility functions. The RL framework proposed in this paper mitigates this problem using the observation that in the real-world DRA systems we analyzed, a major long-term improvement in a system's performance can be achieved by observing the system's state (and making occasional control decisions) at time intervals that are much larger than the resource transfer times. Therefore, each action changes the state of the system instantaneously and deterministically; hence it is sufficient to learn the utility function only over the state space and then simply evaluate various state configurations arising after various possible resource transfer actions, as opposed to using the Q-learning RL algorithm as in [8] where the utility function is learned over the combined state-action space. In the domains where the resource transfer actions do take a long time, it might be possible to come up with a state description where pending actions naturally change the system's state, and so the approach described above can still be used.

As was noted earlier, many different resource allocation methods can be used if the project utility functions $U^i(s)$ are available. So as not to detract from the main objective of this paper (demonstrating the feasibility of learning utility functions), we demonstrate how the RL methodology can work with the following simple resource allocation policy: each project i computes the change in U_i if a unit of resources were added or removed, and resources were then taken away from the least needy project and given to the most needy one as long as the combined benefit to the two projects outweighs the cost of the resource transfer. Several resource units can be transferred during a single time step by recomputing the project utilities after transferring each unit.

If project utility functions are concave increasing (each additional unit of resources brings at most as much benefit as the previous one) and the resource units are infinitely small, then the resource trading approach described above converges to the globally optimal resource allocation that maximizes at every

time step the sum of utilities of all projects. As an outline of the proof, observe that the trading of an infinitely divisible resource stops when $dU_i/dr_i = dU_j/dr_j$ for all i and j – when the marginal benefits of slightly increasing the resource holdings are the same for all projects. This is exactly the necessary condition for global optimality, which can be derived using the method of Lagrange multipliers. This condition is also sufficient when concave increasing utility functions are used with convex resource constraints, such as having a fixed total amount of resources. Since we are not restricting project utility functions to be concave, the above optimality guarantee does not generally hold. However, in practice it is not necessary to trade resources at every time step until the optimal allocation is reached, since the resulting resource allocation might no longer be optimal during the next time step when the system's state will change. Instead, a single resource transfer between the least needy and the most needy project might be sufficient.

Any kind of parameterized utility function approximation architecture can be used in the proposed resource allocation framework, since the only criterion for its compatability with the RL methodology for tuning parameters of utility functions is differentiability of $U(s)$ with respect to each tunable parameter. As will be shown in Section 4, it is sufficient to use an n-dimensional state vector s when n resource types can be shared among projects, with each component of the state vector for a given project being its utilization of the corresponding resource. This leads to a relatively small size of s in our approach, suggesting that a "local" rule-based function approximation architecture can be used (as opposed to a "global" architecture such as a multi-layer perceptron neural network). The main benefits of using rule-based architectures are that the results of using RL for tuning their parameters can be visually inspected and interpreted, and that they can be easily initialized to correspond to "reasonable" utility functions. Their main disadvantage is the exponential increase in the number of rules as the number of input variables increases. A fuzzy rulebase was chosen to represent $U(s)$ in our experiments, and the resulting architecture for dynamic resource allocation based on fuzzy reinforcement learning will be called DRA-FRL.

3.2 Fuzzy Rulebase

A fuzzy rulebase is a function f that maps an input vector $x \in \Re^K$ into a scalar output y. The following common form of the fuzzy rules is used in this paper:

- Rule i: IF (x_1 is S_1^i) and (x_2 is S_2^i) and ... (x_K is S_K^i) THEN ($output = p^i$),

where x_j is the jth component of x, A_j^i are fuzzy categories used in rule i and p^i are the tunable output parameters. The output of the FRB $f(x)$ is a weighted average of p^i:

$$y = f(x) = \frac{\sum_{i=1}^{M} p^i w^i(x)}{\sum_{i=1}^{M} w^i(x)}, \tag{1}$$

where M is the number of fuzzy rules and $w^i(x)$ is the weight of rule i computed as $w^i(x) = \prod_{j=1}^{K} \mu_{A_j^i}(x_j)$, where $\mu_{A_j^i}(x_j)$ is a *membership function* taking values in the interval [0,1] that determines the degree to which an input variable x_j belongs to the fuzzy category A_j^i. A separate rule is used for each combination of fuzzy categories A_j^i, which should jointly cover the space of all possible values that the input vector x can take. Therefore, each parameter p^i gives the output value of the FRB when the input vector x "completely" belongs to the region of the state space described by the fuzzy categories A_j^i of rule i. Since some or all of the fuzzy categories can overlap, several p^i usually contribute to the rulebase output, with their contributions being weighted by the extent to which x belongs to the corresponding regions of space.

If the variable ranges are not known *a priori*, a statistical procedure can first be used to estimate them, and the fuzzy categories can then be defined by splitting the range of each variable into two or more fuzzy categories, depending on the expected complexity of the function being approximated. For example, if the function is expected to be monotonic over the range of some variable, two fuzzy categories will be sufficient, but if the function is expected to be hump-shaped, then three categories will be needed. As the number of relevant input variables and the fuzzy categories used for each variable is gradually decreased, the utility function approximation learned with RL will differ more and more from the true utility function, gradually degrading the performance of the utility-based resource allocation mechanism

described in the previous section.

If the membership functions $\mu()$ are kept constant and only the output coefficients p^i are tuned, then the fuzzy rulebase becomes equivalent to a linear combination of basis functions – a well known statistical regression model. If the membership functions are tuned as well, then the above form of the fuzzy rulebase was proven to be a universal function approximator [13], just like a multi-layer perceptron neural network. In practice, tuning the output coefficients p^i is often sufficient to come up with a good policy, and for simplicity of exposition only such tuning is demonstrated in this paper.

3.3 Reinforcement Learning Algorithm

We first describe the general mathematical context of the Markov Decision Problem (MDP) where reinforcement learning (RL) algorithms can be used. An MDP for a single agent (decision maker) can be described by a quadruple (S, A, R, T) consisting of:

- A finite set of states S
- A finite set of actions A
- A reward function $r : S \times A \times S \rightarrow \Re$
- A state transition function $T : S \times A \rightarrow PD(S)$, which maps the agent's current state and action into the set of probability distributions over S.

At each time t, the agent observes the state $s_t \in S$ of the system, selects an action $a \in A$ and the system changes its state according to the probability distribution specified by T, which depends only on s_t and a_t. The agent then receives a real-valued reinforcement signal $r(s_t, a_t, s_{t+1})$. The agent's objective is to find a stationary policy $\pi : S \rightarrow A$ that maximizes expectation of either a discounted sum of future rewards or the average reward per time step starting from any initial state, which are the two most popular optimization criteria.

The *value* of a state s under a policy π in the discounted reward case is defined as:

$$V^{\pi}(s) = E[\sum_{t=0}^{\infty} \gamma^{t} r(s_t, \pi(s_t), s_{t+1}) | s_0 = s]. \tag{2}$$

Given a policy π, a well-known procedure for iteratively approximating $V^{\pi}(s)$ is called *temporal difference* (TD) learning. Its simplest form is called TD(0):

$$V^{\pi}(s_t) \leftarrow V^{\pi}(s_t) + \alpha_t (r(s_t, \pi(s_t), s_{t+1}) + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t)), \tag{3}$$

where α_t is the learning rate. The above iterative procedure converges to correct value functions as long as the underlying Markov chain of states encountered under policy π is irreducible and aperiodic (the system can transfer from any state to any other state using n, $n+1$, $n+2$, ... time steps for n greater than some value N) and the learning rate α_t satisfies $\sum_{t=0}^{\infty} \alpha_t = \infty$ and $\sum_{t=0}^{\infty} \alpha_t^2 < \infty$ [9]. For example, $\alpha_t = 1/t$ satisfies this conditions.

After a value function $V^{\pi}(s)$ is obtained by executing the above TD algorithm until the state values stop changing noticeably, a "better" policy π' can be obtained (resulting in a larger average reward per time step) by taking "greedy" actions with respect to the values $V^{\pi}(s)$:

$$\pi'(s_t) = \underset{a}{\text{argmax}} E[r(s_t, a, s_{t+1}) + \gamma V^{\pi}(s_{t+1})], \tag{4}$$

where $E[.]$ denotes expected value. The procedure of alternating the policy evaluation and policy improvement steps described above is called *policy iteration*, and it is guaranteed to converge to the optimal policy π^* (a policy π^* is optimal if and only if $V^{\pi^*}(s) \geq V^{\pi}(s)$ for any other policy π and for all states $s \in S$) in a finite number of iterations for a finite state and action space MDP [7]. In practice, it is often not necessary to evaluate a policy until seeming convergence of its value function before changing the policy, and one can use a policy that is always "greedy" with respect to the current value function, which is continually updated by TD learning. Note that due to the fact that a resource transfer action changes the system's state s immediately to a new state $\tilde{s}$ from which the state evolution proceeds, the policy

improvement equation (4) becomes in the DRA domain $\pi'(s_t) = \underset{a}{\operatorname{argmax}}[r(s_t, a, \tilde{s}_t) + V^{\pi}(\tilde{s}_t)]$.

The temporal differencing approach to policy evaluation described by equation (3) is based on assigning a value to each state, which becomes impractical when the state space becomes very large or continuous since visits to any given state become very improbable. In this case, a function approximation architecture needs to be used in order to generalize the value function across neighboring states. Let $\hat{V}(s,p)$ be an approximation to the optimal value function $V^*(s)$ based on a linear combination of basis functions with a parameter vector p: $\hat{V}(s,p) = \sum_{i=1}^{M} p^i \phi^i(s)$, where $p = (p^1, p^2, ..., p^M)^T$ and $\phi(s,a) = (\phi^1(s), \phi^2(s), ..., \phi^M(s))^T$. The parameter updating rule in this case becomes (executed for all parameters simultaneously):

$$\begin{aligned} p^i_{t+1} &= p^i_t + \alpha_t \frac{\partial}{\partial p^i}[r(s_t, \pi(s_t), s_{t+1}) + \gamma \hat{V}(s_{t+1}, p_t) - \hat{V}(\tilde{s}_t, p_t)]^2 \\ &= p^i_t + \alpha_t [r(s_t, \pi(s_t), s_{t+1}) + \gamma \hat{V}(s_{t+1}, p_t) - \hat{V}(\tilde{s}_t, p_t)] \frac{\partial}{\partial p^i} \hat{V}(\tilde{s}_t, p_t) \\ &= p^i_t + \alpha_t [r(s_t, \pi(s_t), s_{t+1}) + \gamma \hat{V}(s_{t+1}, p_t) - \hat{V}(\tilde{s}_t, p_t)] \phi^i(\tilde{s}_t), \end{aligned} \quad (5)$$

where $\tilde{s}_t = s_t$ if no resource transfer takes place. Note that the policy $\pi(s_t)$ does not specify the standard MDP actions that compose MDP policies. Instead, $\pi(s_t)$ only affects the way states are sampled for the TD update. The above iterative procedure is guaranteed to converge to the correct parameter values if certain additional conditions are satisfied [9]. The most important ones are that the basis functions $\phi^i(s)$ are linearly independent and that the states are sampled for update according to the steady-state distribution of the underlying Markov chain for the given policy. No convergence analysis is available so far for the case when states are sampled based on value maximization (as is done in the DRA-FRL framework), but our experimental results suggest that this is a viable practical alternative.

The equation (5) is used for updating parameters p^i of the fuzzy rulebase approximation to the state value function, with $\phi^i(s)$ being the normalized weight of each fuzzy rule i: $\frac{w^i(s)}{\sum_{i=1}^{M} w^i(s)}$. The exact structure of the fuzzy rulebase for the DRA problem is given in the next section.

4 Experimental Setup and Results

The DRA-FRL architecture was evaluated on the problem of migrating CPUs and 1GB memory blocks between resource pools in Solaris 10 in response to a stochastically changing workload in each pool (partition). A pool of 19 CPUs and 22 GB of RAM were allocated among 3 partitions on a Sun Fire (TM) 2900 machine. Threads arrived stochastically into each partition. Each thread was randomly chosen to be CPU-intensive or memory-intensive. A CPU-intensive thread would utilize one CPU by about 70% and use about 60KB of RAM. A memory intensive thread would utilize one CPU by about 1% and use 27MB of RAM. The average length of each thread was at least 10 times larger than the 1 second time interval between state evaluations in each partition, which was chosen in turn to be much larger than the 0.01 second time interval required to perform a resource migration in Solaris 10. A newly generated thread would be dispatched only if enough resources were available for its execution: one CPU for each CPU-intensive thread and 27/0.03 = 900MB of RAM for each memory-intensive thread, as we wanted to keep the total memory utilization in each partition below 3%. Threads that could not be dispatched were queued in a separate queue for each partition. The objective of the DRA experiments described below was to minimize the average queue length among all partitions plus the average resource transfer cost (in cases when a cost was assigned to transferring a resource unit).

One of the benchmarks used for DRA-FRL was the optimal static allocation of resources computed by solving a queuing model for the expected average queue length in each partition as a function of the number of resources in that partition. A discrete optimization problem was then solved to find the allocation of CPUs and memory among partitions that minimized the average queue length among all partitions. If a partition had $n1$ CPUs and X GB of RAM, then the CPU-intensive threads faced a $G/G/n1$ queuing model and the memory-intensive threads faced a $G/G/n2$ queuing model with $n2 = (0.03/27)X$, where the first letter "G" denotes a "general" thread arrival distribution, the second "G" denotes a "general" thread service time distribution, and the third letter "n" denotes the number of parallel "servers" that can process threads. The total expected queue length in such a partition was found by adding expected backlogs in the CPU and memory queuing models.

Unfortunately, the expected queue length of a $G/G/n$ model cannot be computed exactly. In order to allow accurate comparisons of DRA-FRL with the optimal queuing model, the system was simplified by using a deterministic (D) thread service time distribution, making the length of each thread to be 24 seconds. The most accurate approximations of average queue length exist for the case when the thread arrival process is memoryless (M). Therefore, for comparison purposes, threads were generated according to a Poisson process. Let λ be the thread arrival rate and $\mu = 1/24$ be the thread service rate. Following the work in [1], the expected queue length $E[Q_D]$ of an $M/D/n$ queue was approximated using the formula

$$E[Q_D] = E[W_M]\{\frac{(1-\rho)n}{n+1} + \frac{\rho}{2}\}\lambda \tag{6}$$

when $\rho = \frac{\lambda}{\mu n} < 0.7$ and using the formula

$$E[Q_D] = E[W_M]\{\frac{16\rho n + (1-\rho)(n-1)(\sqrt{4+5n}-2)}{32\rho n}\}\lambda \tag{7}$$

when $\rho > 0.7$, where $E[W_M]$ is the expected *waiting time* in an $M/M/n$ queue, which in turn can be computed exactly using the formula

$$E[W_M] = \frac{(n\rho)^n}{n!(1-\rho)n\mu}\{(1-\rho)\sum_{k=0}^{n-1}\frac{(n\rho)^k}{k!} + \frac{(n\rho)^n}{n!}\}^{-1} \tag{8}$$

The following arrival rates for (CPU, memory)-intensive threads were used for the 3 partitions: (0.22, 0.19), (0.18, 0.23), and (0.20, 0.21). With these arrival rates, the optimal fixed resource allocation of (CPUs, GB of RAM) for the 3 partitions was found to be (7,7), (6,8), (6,7), with the corresponding average backlogs among 3 partitions of 1.02, 0.74, 1.99 and the total average backlog of 1.25. When the actual system was observed with the above arrival rates for 40000 time steps, the average partition backlogs were found to be 1.21, 0.99, 2.53 and the total average backlog was found to be 1.58 with a standard deviation of 0.01. The observed backlogs are higher than the predicted ones because of expected deviations of our hardware simulator from the "ideal" queuing system with a Poisson arrival process. However, the relative values of the observed backlogs are very close to those in the "ideal" system,

suggesting that the computed optimal resource allocation is most likely optimal in the real system as well, especially given the integer resource constraints.

Since all partitions received an approximately equal resource allocation in the optimal queuing model, the number of CPUs and GB of RAM in each partition were constrained to be between 4 and 10 for all dynamic allocation policies. When evaluating DRA-FRL, two variables were used as inputs to the fuzzy rulebase when estimating the value of each partition:

- x_1 = CPU utilization in that partition
- x_2 = memory utilization in that partition divided by 0.03.

Other variables were evaluated as well, such as the amount of each resource the project is using or the length of its job backlog, but they did not give any significant improvements in performance.

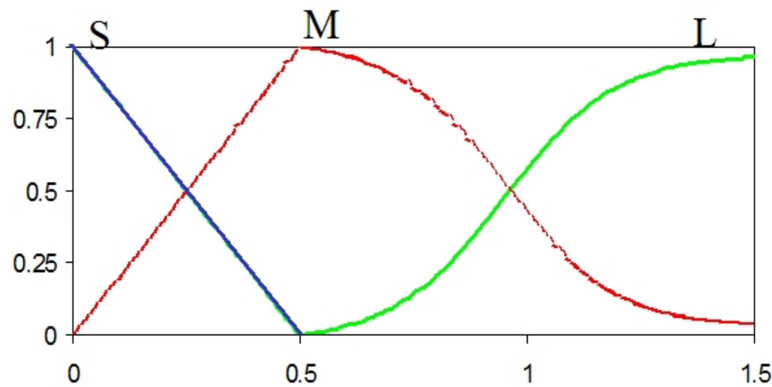

Figure 1. Fuzzy membership functions for each input variables.

Each input variable was divided into three fuzzy categories: Small (S), Medium (M) and Large (L), allowing DRA-FRL to learn nonlinear value functions. Hence, the following 9 fuzzy rules were used:

IF (x_1 is S) and (x_2 is S) THEN (p^1)

IF (x_1 is S) and (x_2 is M) THEN (p^2)

IF (x_1 is S) and (x_2 is L) THEN (p^3)

IF (x_1 is M) and (x_2 is S) THEN (p^4)

IF (x_1 is M) and (x_2 is M) THEN (p^5)

IF (x_1 is M) and (x_2 is L) THEN (p^6)

IF (x_1 is L) and (x_2 is S) THEN (p^7)

IF (x_1 is L) and (x_2 is M) THEN (p^8)

IF (x_1 is L) and (x_2 is L) THEN (p^9)

As was explained in section 3.2, the weight of each rule i is the product of the membership functions used in that rule. The shapes of the three membership functions used for each input variable are shown in Figure 1. For example, the degree to which x_1 is Small is given by the function $\mu_S(x_1)$, while the degree to which x_2 is Large is given by the function $\mu_L(x_2)$. The exact from of the "Large" membership function was $\mu_L(x) = 1/(1+a\exp(-b(x-Mid))) - 1/(1+a)$, where the values of a and b were chosen to be 39 and 8.14, so that 95% of the vertical range of $\mu_L(x)$ would be contained between $Mid = 0.5$ and $Max = 1.4$. The "Medium" membership function for $x > Mid$ was computed as $\mu_M(x) = 1 - \mu_L(x)$. Equation (5) was used for tuning the output parameters p^i of the fuzzy rulebase at every time step.

The objective of each partition was to minimize the future discounted sum of penalties, which were computed at every time step as follows:

$$r(s_t, s_{t+1}) = KB(t+1) + N_{CPU}(t)C_{CPU} + N_{MEM}(t)C_{MEM}, \tag{9}$$

where $B(t+1)$ is the backlog of that partition at time $t+1$, C_{CPU} and C_{MEM} are the *transfer costs* per CPU or 1GB of RAM (their values are important only relative to K, which was set at a reference value of 5), and $N_{CPU}(t)$ and $N_{MEM}(t)$ are the number of CPUs and GB of RAM transferred at time t in or out of this partition. Note that according to Little's Law, the queue length is proportional to the queue waiting time. Thus, minimizing the first component of the above reinforcement signal is equivalent to minimizing the job response time, a common service level objective. The transfer cost can reflect the resource downtime during the migration process (it is negligible in the Solaris environment, but will not be such in a general data center environment) or the amount of other resources consumed in order to enable the transfer.

In addition to the optimal static allocation, performance of the DRA-FRL architecture was also compared against a "reactive" policy, which worked as follows. First, partition i with the longest backlog

was identified. Then, for every resource whose utilitzation was greater than 75% in partition i, a partition j was sought with the smallest utilization of this resource out of all partitions with no backlog. If such a partition was found, then one resource unit was transferred from partition j to partition i. Note that the policy of performing early resource transfers so as to keep equal resource utilization in all partitions can perform very poorly in the general case scenario we are considering where partitions can have different job arrival rates, since partitions with a higher job arrival rate will have a higher chance of forming a backlog of waiting jobs.

Since the DRA scenario is based on a queuing model for arriving threads and since the thread arrivals are stochastic, many time steps are required in order for any learning architecture to correlate the reinforcement received from various decisions *in each region of the state space*. For any combination of problem parameters, the DRA-FRL architecture was first in the tuning mode for 40000 time steps (each 1 second long) and its performance was then evaluated for 40000 more time steps. Performance of the policy was defined as the average cost per time step obtained by the policy. The standard deviation of performance for the 40000 testing time steps was always less than 2% of the performance itself. Performance of the reactive policy was also averaged over 40000 time steps.

For demonstration purposes, we present results of using only one policy iteration step described by equations (3) and (4), with the "reactive" policy being used as the initial policy. In practice, once the parameter vector p stops changing noticeably during the policy evaluation phase, the algorithm can automatically switch from executing the reactive migration policy to executing the utility-based migration policy.

We have also tried using a "greedy" policy with respect to the continually updated value function, but observed an "arms race" taking place in our multi-agent domain: a situation where one partition temporarily experiences a very high load and updates its value function to place a higher value on additional resources. As a result, this partition temporarily gets a higher fraction of them than other partitions, which consequently start observing higher backlog costs and hence also adjust parameters of their value functions so as to place more value on additional resources. As the "arms race" continues, partitions can converge to a highly suboptimal game theoretic equilibrium. Dealing with this problem is

Table 1. Average cost per time step of the considered resource allocation policies

	Optimal Fixed Allocation	Reactive Policy	Utility-based Policy
Transfer Cost = 0	7.9	2.12	1.56
Transfer Cost = 0.31	7.9	2.36	1.99

left for the future work, and in the current paper we use the more time demanding approach of performing the policy evaluation phase completely before changing the policy.

The experimental results are summarized in Table 1. In the first set of experiments (first row), both C_{CPU} and C_{MEM} were set to 0. The utility-based policy learned by observing the reactive policy outperforms the reactive policy by 26%. The second set of experiments evaluated the adaptability of the DRA-FRL architecture to the presence of a known resource transfer cost. We observed that in the first set of experiments the utility-based policy performed on average 1.3 resource transfers per time step while the reactive policy performed on average 0.39 resource transfers per time step. Hence, we calculated that if C_{CPU} and C_{MEM} were both set to 0.31 and the DRA-FRL architecture could not adapt to the presence of the resource transfer costs and still performed on average 1.3 resource transfers per time step, its performance would match that of the reactive policy. However, the second set of experiments (second row) shows that the DRA-FRL architecture does adapt to the presence of a known resource transfer cost: the utility-based policy reduces the average number of resource transfers per time step down to 0.67 (making only those transfers whose benefit outweighs the transfer cost) and still outperforms the reactive policy by 16%.

5 Future work

A possible approach to avoiding the "arms race" in the context of multi-agent reinforcement learning was presented in [10]. The key idea there was to add a new state variable to each agent, encoding the current demand for the common resource (channel bandwidth in that paper) from all other agents in the system. This allowed agents to learn to "back off" gradually from using the common resource if the demand from other agents is larger than a certain threshold. This resulted in agents taking turns in using

large portions of the common resource according to their stochastically evolving needs, as opposed to all of them always trying to get as much of the resource as possible regardless of their current need. In the DRA setting, the combined demand for the common pool of CPUs and memory can be represented as the average state value of all other agents. If this value is high, then other agents expect to observe a high backlog cost in the near future and hence are in a greater need for resources. We plan to evaluate this and other approaches to avoiding the "arms race" during dynamic resource allocation in our future work.

Another possible extension of the present work is to allow each partition to tune the structure of the fuzzy membership functions they use. This will make the value functions more flexible and will result in a better potential match to the optimal value functions. Such a tuning can be easily done for the "large" membership functions used in this paper using equation (5), since the rule output is differentiable with respect to the membership function parameters a and b.

6 Acknowledgement

The author would like to thank Declan Murphy from Sun Microsystems for helpful comments and corrections to this paper.

References

[1] F. Bartelo, J. Paradells. "Performance Evaluation of Public Access Mobile Radio (PAMR) Systems with Priority Calls," In Proceedings of the 11th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications, September 18-21, 2000.

[2] D. Bertsekas and J. Tsitsiklis, *Neuro-Dynamic Programming*, Athena Scientific, 1996.

[3] A. Byde, M. Salle and C. Bartolini, "Market-Based Resource Allocation for Utility Data Centers," HP Technical Report HPL-2003-188. http://www.hpl.hp.com/techreports/2003/HPL-2003-188.pdf

[4] T. Jaakkola, M. I. Jordan, and S. P. Singh, "On the convergence of stochastic iterative dynamic programming algorithms." *Neural Computation*, Vol. 6, No. 6, 1994, pp. 1185–1201.

[5] L. P. Kaelbling, L. M. Littman, and A. W. Moore, "Reinforcement learning: a survey." *Journal of Artificial Intelligence Research*, Vol. 4, 1996, pp. 237–285.

[6] "Solaris Containers." A Sun Microsystems white paper. Available electronically at http://wwws.sun.com/software/whitepapers/solaris10/ grid_containers.pdf

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

[8] G. Tesauro, R. Das, W. E. Walsh, and J. O. Kephart, "Utility-function-driven Resource Allocation in Autonomic Systems," in Proceedings of the 2nd IEEE International Conference on Autonomic Computing (ICAC-05), 2005.

[9] J. N. Tsitsiklis and B. Van Roy, "An Analysis of Temporal-Difference Learning with Function Approximation," *IEEE Transactions on Automatic Control*, Vol. 42, No. 5, May 1997, pp. 674-690.

[10] D. Vengerov, N. Bambos and H. R. Berenji. "A Fuzzy Reinforcement Learning Approach to Power Control in Wireless Transmitters." *IEEE Transactions on Systems, Man, and Cybernetics B*. Vol. 35, Issue 4, August 2005.

[11] D. Vengerov, N. Iakovlev. "A Reinforcement Learning Framework for Dynamic Resource Allocation: First Results," in Proceedings of the 2nd IEEE International Conference on Autonomic Computing (ICAC-05), 2005.

[12] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das, "Utility functions in autonomic systems," In proceedings of International Conference on Autonomic Computing, 2004. http://www.research.ibm.com/people/w/wwalsh1 /Papers/icac04NeDAR.pdf

[13] L.-X. Wang, "Fuzzy systems are universal approximators," In Proceedings of the IEEE International Conference on Fuzzy Systems (FUZZ-IEEE '92), 1992, pp. 1163-1169.