A Reinforcement Learning Framework for Online Data Migration in Hierarchical Storage Systems *

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

Abstract

Multi-tier storage systems are becoming more and more widespread in the industry. They have more tunable parameters and built-in policies than traditional storage systems, and an adequate configuration of these parameters and policies is crucial for achieving high performance. A very important performance indicator for such systems is the response time of the file I/O requests. The response time can be minimized if the most frequently accessed ("hot") files are located in the fastest storage tiers. Unfortunately, it is impossible to know a priori which files are going to be hot, especially because the file access patterns change over time. This paper presents a policy-based framework for dynamically deciding which files need to be upgraded and which files need to be downgraded based on their recent access pattern and on the system's current state. The paper also presents a reinforcement learning (RL) algorithm for automatically tuning the file migration policies in order to minimize the average request response time. A multi-tier storage system simulator was used to evaluate the migration policies tuned by RL, and such policies were shown to achieve a significant performance improvement over the best hand-crafted policies found for this domain.

Keywords: Self-Optimizing Systems, Markov Decision Process, Reinforcement Learning, Cost Functions, Data Migration, Multi-Tier Storage, Fuzzy Rulebase.

1 Introduction

The computer industry is reaching a consensus that large scale systems should be self-managing and self-optimizing. Otherwise, too many human experts would be needed to manage them, and their performance would still be inadequate because of constant unpredictable changes in external environment. Most of the work done toward achieving these goals has focused on the self-management objective,

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

which is much easier to achieve. The current industry-standard policy-based management approaches use built-in rules (or local agents) that specify what needs to be done when certain conditions arise. These rules takes actions that improve the situation, but since they are specified heuristically and the environment around the system evolves over time, it is very difficult to make claims that such built-in rules actually optimize the system's behavior.

Another approach uses the concept of utility functions, representing the desired performance objectives. Within this approach, in order to enable the self-optimization capability, a utility function is formed for representing the desired performance objectives, and the system then automatically takes actions that optimize the current utility function. This approach raises the level of abstraction at which humans get involved into system management: instead of specifying what actions the system should take, human administrators specify/adjust the optimization goals, which the system automatically tries to achieve. In order to implement this approach, however, the administrators need to decide how to form the utility function for each automated decision-making agent responsible for implementing a certain policy. The optimization objective should ideally account for what is likely to happen in the future, rather than just optimizing the current state of affairs, as is done in classical linear and nonlinear programming optimization.

Reinforcement Learning (RL) is an emerging solution approach for making decisions based on statistical estimation and maximization of expected *long-term* utilities (e.g., [12]). Some researchers have recently demonstrated how RL can be used for learning utility functions in real-world dynamic resource allocation problems [1, 17]. Inspired by these works, this paper gives a demonstration of how RL can be used for learning utility functions in the problem of dynamic data migration in hierarchical storage systems. Many important details need to be worked out in order to apply RL to any large-scale real-world problem. The following issues are addressed in this paper: deciding what intelligent agents the system should contain, choosing the variables to be used as inputs to the utility function of each agent, deciding how the outputs of utility functions should be translated into actual data migration decisions, choosing the computational architecture for approximating utility functions, specifying the form of the RL algorithm to be used for updating different parameters of the utility approximation architectures and deciding when that algorithm should be invoked. With respect to the algorithm used, this paper extends an earlier technical report published by the author on this topic [18] by presenting a novel RL-based algorithm for updating parameters of the basis functions $\phi^i(x)$ that are used to approximate the utility function: $\hat{U}(x,p) = \sum_{i=1}^{L} p^i \phi^i(x)$ (Section 4.5).

2 Domain Description

Current enterprise systems store petabytes of data on various storage devices (fast disks, slow disks, tapes) interconnected by storage area networks (SANs). As more data needs to be stored, new storage devices are connected to such networks. The currently available storage devices range widely in price and performance. For example, accessing a file on a new tape drive requires a robot to physically change the tape drives before the new data can be accessed, which can take several seconds (while data on disks can be accessed in milliseconds). For performance reasons, it is desirable to keep the most frequently accessed "hot" data on fast disks while keeping the old "cold" data on the inexpensive slow disks or tapes. Figure 1 shows an example of a multi-tier storage system.

Unfortunately, the file access patterns change over time, and the data layout cannot be optimized once and for all. For example, in e-mail servers the file access frequency tends to decrease as files stay longer

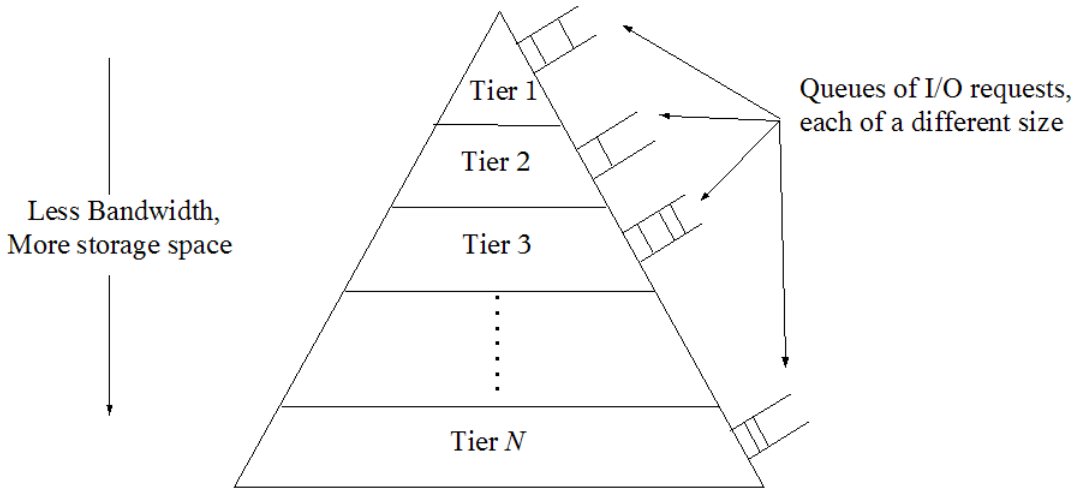

Figure 1. Multi-tier Hierarchical Storage System.

in the system. The file access patterns in commercial Data Centers and Grids depend greatly on the clients connected to such systems at each point in time. Thus, online data migration has the potential to significantly increase performance of computing systems with hierarchical storage.

Recently developed storage technologies (e.g., [8, 13]) allow transparent migration of files between storage tiers without affecting the running applications that are accessing them. However, the data migration policies they use are still based on some simple heuristics and no formal mathematical framework for minimizing the future I/O request response time has been developed. The Aqueduct system [6] uses a control-theoretic framework for dynamically adjusting the rate of background data migration to avoid Quality of Service (QoS) violations for the running applications. However, this system assumes that the data migration plan is given to it as an input. The STEPS architecture [19] for life cycle management in distributed file systems also uses feedback control to dynamically adjust the data migration rate. The migrations in that system are triggered by heuristic threshold-based policies such as *IF space utilization in PREMIUM POOL is greater than 80%, THEN migrate .tmp files greater than 1MB ordered by size to IDLE POOL until the space utilization of PREMIUM POOL is 60%*.

A number of online data migration approaches have been proposed in the research community. However, the approaches we are aware of are reactive in nature: they suggest migrating "hot" files that have been accessed more frequently in the recent past to the faster storage tiers while swapping out the "coldest" files. As a representative example, [11] suggests estimating the expected cost of accessing each file based on the file location and on its estimated access frequency. A file is then upgraded to a faster tier of storage (e.g., from disk to a local cache) if the total cost of all files involved in this migration (such as those that need to be dropped from the cache) is reduced. The cost of accessing each file is dynamically estimated using a sliding average of accesses to similar objects in the recent past.

While reactive approaches such as the one in [11] can definitely bring some benefit to multi-tier storage systems, they do not address several important system-level considerations. For example, reactive approaches do not consider the impact of migration overhead on the future response times of other files in the affected storage tiers, even though a constant migration overhead added to an overloaded storage tier can significantly increase its queuing delays. Therefore, the total benefit received by the transferred files needs to be weighed against the future impact of migration overhead in the particular states of the

affected storage tiers. The future response times of other files are also affected by changes in the file size distribution in each storage tier that result from file migrations. For example, the expected response times of all files in a storage tier will increase if a very large file is migrated into this tier, since the I/O requests to the original files will have to be queued for a long time while an I/O request for the very large file is served. Therefore, faster storage tiers have a preference for "hot" AND small files, and these two properties need to be traded off against each other on a case-by-case basis depending on the particular files and the states of the storage tiers involved in the migration.

This paper presents a methodology for addressing the above tradeoffs by estimating the *cost function* for each storage tier – its expected *future* response time as a function of its current state. Given the cost functions for all storage tiers, the true impact of each migration decision can be easily estimated as the total change to the cost functions of the affected storage tiers that will result if the considered migration were to take place. This approach is consistent with the current industry trends of developing self-optimizing systems, which make decisions that maximize a utility function rather than just following some built-in non-adaptive policies.

3 Hierarchical Storage System Model

Consider a multi-tier storage system as shown in Figure 1. Tier i has a limited bandwidth B_i, which is shared between file I/O requests and file transfers between the tiers (data migration). Multiple copies of a single file can be present in the storage system. Read/write/update requests arrive stochastically for each file. We assume that a certain policy already exists for routing new file write requests (a possible heuristic is to write the file to the fastest tier that has enough available space to fit this file). A file update request is routed to all tiers containing this file. A file read request is routed to the tier which has the smallest expected wait time out of the tiers containing this file. Requests at each tier form a queue and wait for their turn to be processed. The processing time for a read/write/update request for file k in tier i is the size of file k divided by B_i.

File migrations can either be performed proactively or they can be triggered by I/O requests. The latter approach imposes a smaller load on the system, since the migration process in general contains the operation that would need to be executed while serving the I/O request (e.g., reading a file from some storage or writing a file to it). This consideration was found to be very important in our simulations of heavily loaded systems, since a small increase in the load (number of I/O operations that need to be executed) significantly increases the request response time in such systems. Therefore, the RL methodology will be applied in this paper to the I/O-triggered migration policies.

When a file update request is received by some tier, the system decides whether this request should be routed instead to the next fastest tier, creating a new copy of the file in that tier and erasing the old copy present in the original tier. This consideration is made only if the file is not already present in the next fastest tier. When a file read request is received, the system first reads the file from disk and then decides whether this file should be upgraded to the next fastest tier. If an upgrade decision is made and not enough space is available in the faster tier, then some files are migrated out of that tier into the original tier. The migration-induced read/write operations are implemented as file requests that are sent to the corresponding tiers (that is, file migration operations do not take precedence over the regular file I/O requests).

The decision to upgrade a file increases the queue length of the new tier and may also increase the queue length of the old tier if some large files need to be swapped out of the new tier. However, such a

decision may place the overall system into a more desirable state from the point of view of the expected future response time, if the file being upgraded is "hot" enough and the files being swapped out (if any) are "cold" enough. The objective is to design an automated policy for upgrading files so as to minimize the average system response time (possibly divided by the size of each request). The need for an automated migration policy extends to many different implementations of a multi-tier storage system, and some of the mechanisms described above can be easily modified without changing the spirit of the problem.

4 Solution Methodology

4.1 Defining and Using Cost Functions

This section addresses the first three issues (as was mentioned in the Introduction) that need to be resolved when applying RL to a real-world problem: deciding what intelligent agents the system will contain, choosing the variables to be used as inputs to the utility function of each agent, and deciding how the outputs of utility functions should be translated into actual data migration decisions.

Since storage tiers generally possess different characteristics and can be dynamically added or removed from a multi-tier storage system, it is impractical to define a single utility function, mapping each possible data distribution among the tiers and properties of all the tiers into the expected future response time of I/O requests. Instead, we propose a scalable approach of viewing each storage tier as a separate decision-making entity (agent), allowing any pair of such agents to decide whether they want to redistribute the files among themselves at any point of time.

In order to make data migration decisions, each storage tier (agent) learns its own *long-term cost function* $C(s)$, which predicts the future average request response time divided by the file size (which will be called average weighted response time, AWRT), starting from a state s for the tier. An adequate state description is very important for learning such a function: the learning time for a given accuracy is exponential in the number of variables used to describe the state (so fewer variables are preferred), but at the same time all major influences on the future cost should be included so as to predict more accurately the future evolution of the agent's state. Some methods are proposed in [7] for dynamically choosing the most appropriate set of features that an agent should use in making its decisions, trading off compactness vs. completeness. Such investigations, however, are outside of the scope of this work.

The state vector s we used for each tier is based on the notion of "temperature" of each file, which approximates its access rate. In order to compute this estimate, the system keeps track of read/update requests for each file and uses these statistics to predict the probability of that file being accessed in the near future. More precisely, whenever a file k is accessed at time t, the system updates an estimate of the request interarrival time τ_k for this file:

$$\tau_k = \alpha\tau_k + (1-\alpha)(t - L_k), \tag{1}$$

where L_k is the last access time for file k and $\alpha < 1$ is a constant that determines how fast the past information should be discounted. The temperature T_k of file k is computed as $T_k = 1/\tau_k$. The following state variables were then formed:

- s_1 = average temperature of all files stored and queued (those for which the write requests have already been received) in the tier. This variable predicts the number of accesses per unit of time

the tier will receive in the future. Note that if requests are infrequent and queuing does not occur, AWRT for each request to tier i is $1/B_i$. Thus, a higher value of s_1 suggests a higher probability of queuing and thus a larger expected cost $C(s)$.

- s_2 = average weighted temperature of all files stored and queued in the tier (temperature multiplied by the file size). This variable predicts the amount of data that will be requested from the tier per unit of time in the future. For a given value of s_1, a higher value of s_2 implies that larger files become relatively hotter, which suggests longer expected queuing times for requests to smaller files (many requests for small files need to wait for a long time while a single request for a large file is being processed) and hence a larger expected cost $C(s)$.

- s_3 = current queuing time in the tier (wait time for a new request to start getting processed). This variable directly correlates with the queuing time for requests that arrive in the near future.

The first two variables are likely to evolve gradually in real storage systems with many files per tier, and hence can be computed at regular time intervals that are large enough to allow for a noticeable change in these variables (e.g., a 1% change). Whenever a read/update request is received for file k that is not already in the fastest storage tier, the system decides whether this file should be upgraded to the next fastest tier, as described in Section 3. In order to make such a decision between tiers i and j, each of them computes its long term cost $C_{not} = C$(state s if no migration is to take place) and $C_{up} = C$(state $\tilde{s}$ that would result after file k is upgraded). The file is then upgraded if

$$C^i_{up} \cdot \tilde{s}^i_1 + C^j_{up} \cdot \tilde{s}^j_1 < C^i_{not} \cdot s^i_1 + C^j_{not} \cdot s^j_1, \tag{2}$$

where s^i_1 is the average temperature of all files in tier i if no migration is to take place and $\tilde{s}^i_1$ is the same quantity if the file in question were to be upgraded. The predicted average response time of a tier (cost) is multiplied by the tier temperature (s_1) in the above migration criterion because AWRT for the multi-tier storage system can be computed as a weighted sum of AWRTs for all tiers, with the weights being the number of requests each tier has served, which is estimated by s_1.

Note that once the tier cost functions $C(s)$ are learned, any file reconfiguration decisions among the tiers can be evaluated using the criterion in equation (2). This approach is robust to tiers being added or removed from the system, is more scalable than centralized approaches and does not have a single point of failure inherent in centralized approaches.

4.2 Fuzzy Rulebase Representation of Cost Functions

Any parameterized function approximation architecture can be used for representing the tier cost functions $C(s)$ and then tuning them with RL. We will use a fuzzy rulebase (FRB) for this purpose because its structure and parameters can be easily interpreted, as will be explained below. An FRB 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 A^i_1) and (x_2 is A^i_2) and ... (x_K is A^i_K) THEN ($output = p^i$),
where x_j is the jth component of x, A^i_j 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:

$$y = f(x) = \frac{\sum_{i=1}^{M} p^i w^i(x)}{\sum_{i=1}^{M} w^i(x)}, \tag{3}$$

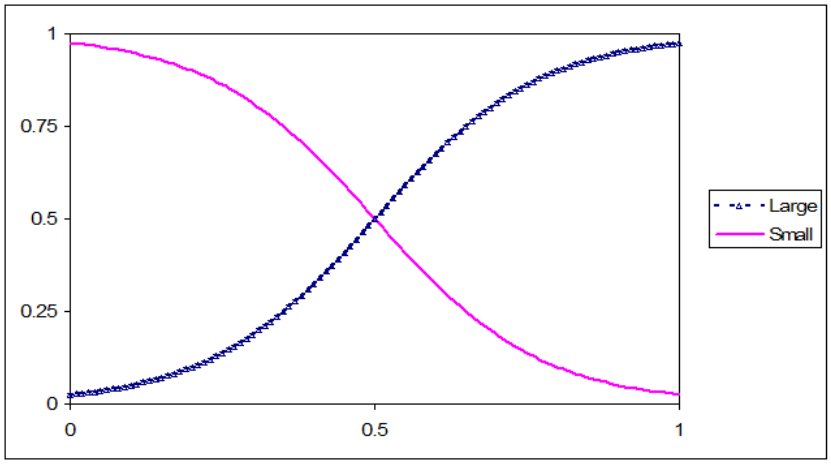

Figure 2. Membership functions showing the degrees to which each input variable is small and large.

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. In contrast, the categories of a "crisp" rulebase (traditional expert system) do not overlap, and hence its output does not change as x moves within each region while changing abruptly as x crosses the border into another region. Such discontinuities are very undesirable when parameter optimization needs to be conducted, which explains the prevalence of fuzzy rulebases in modern industrial control systems.

Figure 2 shows an example of two fuzzy categories (SMALL and LARGE) covering the range of an input variable x_j that takes values in the interval [0,1], with their corresponding membership functions actually being shown. As x_j increases from 0 to 1, the degree to which it is SMALL monotonically decreases, and correspondingly the weight of all fuzzy rules where (x_j is SMALL) also monotonically decreases, while the weight of all fuzzy rules where (x_j is LARGE) monotonically increases. Thus, the output of the FRB also changes monotonically. If one suspects that the function $f(x)$ should trace a hump-shaped form as x_j increases from 0 to 1, then three fuzzy categories should be used for covering the jth dimension, with the output parameters in the fuzzy rules where (x_j is MEDIUM) being larger than those where (x_j is SMALL) or (x_j is LARGE). If sufficiently many fuzzy categories are used in each input dimension, with the membership functions being appropriately defined, then the FRB described above can approximate arbitrarily well any continuous function [20].

The simplest possible (but still reasonably descriptive) FRB would use only two membership functions for each input variable x_j, one describing the degree to which x_j is SMALL ($\mu_{S_j}(x_j)$) and the other describing the degree to which x_j is LARGE ($\mu_{L_j}(x_j)$). Since the goal of this paper is to *demonstrate*

the possibility of using RL for automatically adapting file migration policies, we will use the simple version of the FRB suggested above. To further simplify the matters, we constrain all input dimensions to use membership functions of the same form: $\mu_{L_j}(x_j) = 1/(1 + a_j e^{-b_j x})$ and $\mu_{S_j}(x_j) = 1 - \mu_{L_j}(x_j)$, as shown in Figure 2. The S-shape form of the membership functions is useful when the ranges of x_j are not constrained and very large or very small values can sometimes be observed, which may be the case with the state variables defined in Section 4.1.

Each agent (storage tier) will store its own parameter values p^i and whenever a migration decision needs to be made, the agent will compute the costs $C(s)$ and $C(\tilde{s})$ described in Section 4.1 by evaluating each rule in its FRB at the corresponding input value of $x = (s_1, s_2, s_3)$ or $x = (\tilde{s}_1, \tilde{s}_2, \tilde{s}_3)$ and then combine the output parameters p^i of these rules using equation (3).

4.3 Reinforcement Learning Methodology for Tuning Cost Functions

The more accurately the cost function $C(s)$ of each tier represents the expected future response of its I/O requests starting from each state s, the more beneficial will be the migration decisions made using equation (2). The cost optimization framework proposed in this paper uses Reinforcement Learning (RL) for tuning the parameter values p^i of the fuzzy rulebase used by each tier for approximating its actual future costs per time step. The RL theory was originally developed for Markov Decision Processes (MDPs), where the agent's state evolves as a Markov chain conditional on its current state and the action taken in that state. However, each agent in our domain has to make decisions in a Partially Observable Markov Decision Process (POMDP), where partial observability refers to the fact that the state vector s described in Section 4.1 does not contain all relevant information for predicting its future evolution even if the tier cost functions remain fixed. For example, knowing the location and the current "temperature" of each file in the system would definitely help making such a prediction. However, constructing cost functions that make use of such information is clearly infeasible, as it will require the state vector s to have an enormous number of dimensions.

A good overview of methods for learning the long-term cost functions in POMDPs is given in [2]. The main difference among the existing approaches is the extent to which the agent uses internal memory to store information about past events, which supplements the current state observation for predicting the future evolution of the system. Some of the memory-based approaches include Utile Suffix Memory [7], recurrent neural networks [5], and finite state controllers [9]. However, some researchers have also reported that memoryless approaches, where the agent uses only the current observation, can also give good results in practice. Since the goal of this paper is to provide a framework for managing multi-tier storage systems using RL (rather than filling this framework with the best possible algorithms), a simple memoryless approach will be presented in this paper, where each agent uses only the observation vector described in Section 4.1 to predict the long term cost of its tier. That is, the agent simply assumes that it acts in an MDP, and the observation vector is treated as the state vector. The experimental results in Section 6 show that even this simple approach can still give very good results.

In order to accumulate sufficient statistics about the request response times in each state, a storage tier updates the parameters of its cost function only when k or more I/O requests have been processed since the last migration decision. This condition gets triggered at irregular time intervals, requiring a continuous time description of the updating process. A continuous time MDP (also known as a Semi-Markov Decision Process, SMDP) for a single agent can be described as follows. Given a set of states S and a set of actions A, the nth decision for the agent is to choose an action $a_n \in A$ in the state $s_n \in S$.

The next state s_{n+1} is then chosen according to the transition probability $P(s_{n+1}|s_n, a_n)$. Once the state s_{n+1} is chosen, the time τ_n the agent spends in state s_n is sampled from the probability distribution $F(\tau|s_n, a_n, s_{n+1})$. Finally, the agent accumulates cost in state s_n at the rate $\rho(s_n, a_n)$. The agent's objective is to find a stationary policy $\pi : S \rightarrow A$ that minimizes $E[\int_0^\infty e^{-\beta t}\rho(s(t), a(t))dt]$, which is the expected long-term cost obtained as the agent's state evolves over time, where β is a discounting factor.

The *cost-to-go* of a state s under a policy π is defined as the sum of future costs that the agent expects to obtain starting from state s, with costs further out into the future being discounted to a larger extent so as to represent a greater uncertainty the agent has about them:

$$C^\pi(s) = E[\int_0^\infty e^{-\beta t}\rho(s(t), a(t))dt \quad |s(0) = s], \tag{4}$$

where E[] is the expected value operator. Given a policy π, a well-known procedure for iteratively approximating $C^\pi(s)$ is called *temporal difference* (TD) learning [12]. While this procedure was originally developed for MDPs, our experimental results demonstrate that it can also be successfully applied to SMPDs. Let c_n be the total cost the agent has accumulated in state s_n until its transition to s_{n+1}. The following form of TD(λ) can then be used, which is executed simultaneously for all $s \in S$ whenever a state transition occurs:

$$\hat{C}^\pi(s) \leftarrow \hat{C}^\pi(s) + \alpha_n(c_n + e^{-\beta\tau_n}\hat{C}^\pi(s_{n+1}) - \hat{C}^\pi(s_n))z_n(s), \tag{5}$$

where $\hat{C}^\pi(s)$ is the current approximation to $C^\pi(s)$, α_n is the learning rate at the nth state transition, τ_n is the time the agent spends in state s_n, and $z_n(s)$ is the *eligibility trace* of each state s, which is based on the number of times that state has been visited in the recent past and therefore determines the extent to which each state is "responsible" for what has just happened. Correspondingly, the cost-to-go of states that are more responsible gets updated to a larger extent. The eligibility trace is initialized to 0 and is then simultaneously updated for all $s \in S$ as follows: $z_n(s) = \lambda e^{-\beta\tau_n}z_{n-1}(s)$ if $s \neq s_n$ and $z_n(s) = \lambda e^{-\beta\tau_n}z_{n-1}(s) + 1$ if $s = s_n$. When $\lambda = 0$, only the cost-to-go of the most recently visited state gets updated. As λ approaches 1, the agent retains more "memory" of what has happened in the past and updates the states according the total number of visitations over a progressively longer time history.

TD(λ) has been proven to converge for MDPs to the correct cost-to-go function C^π 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 m, $m+1$, $m+2$, ... time steps for m greater than some value M) and the learning rate α_n satisfies $\sum_{n=0}^\infty \alpha_n = \infty$ and $\sum_{n=0}^\infty \alpha_n^2 < \infty$ [14]. For example, $\alpha_n = 1/n$ satisfies this conditions. The same convergence guarantee holds for the "well-behaved" SMDPs, where pathologies such as an infinite number of state transitions during a finite period of time or an infinite cost accumulated in a given state cannot happen.

After a cost-to-go function estimate $\hat{C}^\pi$ stops changing noticeably in the course of executing the TD algorithm in equation (5) and is deemed to be a good approximation to C^π, then a "better" policy π' (such that $C^{\pi'}(s) \leq C^\pi(s)$ for all $s \in S$) can be obtained by taking "greedy" actions with respect to the costs $\hat{C}^\pi(s)$:

$$\pi'(s_n) = \underset{a}{\text{argmin}} E[c(s_n, a) + e^{-\beta\tau_n}\hat{C}^\pi(s_{n+1})], \tag{6}$$

where $c(s_n, a)$ is a random variable indicating the cost obtained in state s_n if the action a is taken. The procedure described above of alternating the policy evaluation and policy modification steps is called *policy iteration*, and it is guaranteed to converge in MDPs to the optimal policy π^* (a policy π^* is

optimal if and only if $C^{\pi^*}(s) \leq C^{\pi}(s)$ for any other policy π and for all states $s \in S$) in a finite number of iterations for finite state and action space MDPs [3]. In practice, it is often not necessary to evaluate a policy until seeming convergence of its cost-to-go function before changing the policy, and one can use a policy that is always "greedy" with respect to the current cost-to-go function, which is continually updated by TD learning. Note that when the effect of each action gets immediately reflected in the system's state (which happens when the state variables described in Section 4.1 are used and the action is a particular file migration), it is sufficient to compare just the state costs following each possible decision:

$$\pi'(s_n) = \underset{a}{\operatorname{argmin}} E[\hat{C}^{\pi}(f(s_n, a))], \tag{7}$$

where $s_{n+1} = f(s_n, a)$. This idea is used in the migration criterion presented in equation (2). A more detailed discussion of this is given in Section 4.6.

4.4 Using RL in Large State Spaces

The temporal differencing approach to policy evaluation described by equation (5) is based on assigning a cost 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 cost-to-go function across neighboring states. Let $\hat{C}(s, p)$ be an approximation to the optimal cost-to-go function $C^{\pi^*}(s)$ based on a linear combination of basis functions with a parameter vector p: $\hat{C}(s, p) = \sum_{i=1}^{L} p^i \phi^i(s)$. The parameter updating rule in this case becomes for $\lambda < 1$ (executed for all parameters simultaneously):

$$p^i_{n+1} = p^i_n + \alpha_n [c_n + e^{-\beta \tau_n} \hat{C}(s_{n+1}, p_n) - \hat{C}(s_n, p_n)] z^i_n, \tag{8}$$

where the eligibility trace z^i_n is initialized as $z^i_0 = 0$ and is updated as follows: $z^i_{n+1} = \lambda e^{-\beta \tau_n} z^i_n + \frac{\partial}{\partial p^i} \hat{C}(s_{n+1}, p_{n+1}) = \lambda e^{-\beta \tau_n} z^i_n + \phi^i(s_{n+1})$. The above iterative procedure is also guaranteed to converge for MDPs if certain additional conditions are satisfied [14]. The most important ones are that the basis functions $\phi^i(s)$ are linearly independent and that the states for update are sampled according to the steady-state distribution of the underlying Markov chain for the given policy π. Once parameters of the long term cost functions $\hat{C}$ begin to converge, the policy improvement step detailed in equation (7) can be carried out.

An FRB, as described in Section 4.2, is an instance of a linear parameterized function approximation architecture, where the normalized weight $\frac{w^i(s)}{\sum_{i=1}^{M} w^i(s)}$ of each rule i is a basis function $\phi^i(s)$. The cost signal c_n reflects the average weighted response time of all requests served by the storage tier between the state s_n at time t_n and the state s_{n+1} at time t_{n+1}: $c_n = (1/X_n) \sum_{i=1}^{X_n} r_i e^{-\beta(t_{n,i} - t_n)}$, where $t_{n,i}$ is the arrival time of the ith request in state n, r_i is its response time, and X_n is the total number of requests served in state s_n.

4.5 A Novel Methodology for Updating Fuzzy Membership Functions

In addition to tuning parameters p^i it is also possible to tune parameters of the basis functions $\phi^i(s)$. While this is theoretically possible, we are not aware of any work that attempted to do that in the context of function approximation architectures consisting of a linear combination of basis functions. This section demonstrates how the tuning of the parameters a_j (see Section 4.2 for a definition of a_j) can

be performed. The updating procedure is similar to the one described in equation (8), where the partial derivatives $\frac{\partial \hat{C}}{\partial a_j}$ are used in the eligibility trace of a_j (the exact procedure is described in Section 6). Note that in the case described in Section 4.2, where the range of each variable is covered by only two membership functions, $\sum_{i=1}^{M} w^i(s) = 1$ for all $s \in S$. Using this observation, after some algebra, the following expressions were obtained for $\frac{\partial \hat{C}}{\partial a_j}$ for the case of the fuzzy rulebase with 3 input variables, as suggested in Section 4.1:

$$\begin{aligned}\frac{\partial \hat{C}}{\partial a_1} = \quad & \frac{\partial \mu_{S_1}}{\partial a_1}(p_1\mu_{S_2}\mu_{S_3} + p_2\mu_{S_2}\mu_{L_3} + p_3\mu_{L_2}\mu_{S_3} + p_4\mu_{L_2}\mu_{L_3}) \\ & + \frac{\partial \mu_{L_1}}{\partial a_1}(p_5\mu_{S_2}\mu_{S_3} + p_6\mu_{S_2}\mu_{L_3} + p_7\mu_{L_2}\mu_{S_3} + p_8\mu_{L_2}\mu_{L_3}),\end{aligned} \tag{9}$$

$$\begin{aligned}\frac{\partial \hat{C}}{\partial a_2} = \quad & \frac{\partial \mu_{S_2}}{\partial a_2}(p_1\mu_{S_1}\mu_{S_3} + p_2\mu_{S_1}\mu_{L_3} + p_3\mu_{L_1}\mu_{S_3} + p_4\mu_{L_1}\mu_{L_3}) \\ & + \frac{\partial \mu_{L_2}}{\partial a_2}(p_5\mu_{S_1}\mu_{S_3} + p_6\mu_{S_1}\mu_{L_3} + p_7\mu_{L_1}\mu_{S_3} + p_8\mu_{L_1}\mu_{L_3}),\end{aligned} \tag{10}$$

$$\begin{aligned}\frac{\partial \hat{C}}{\partial a_3} = \quad & \frac{\partial \mu_{S_3}}{\partial a_3}(p_1\mu_{S_1}\mu_{S_2} + p_2\mu_{S_1}\mu_{L_2} + p_3\mu_{L_1}\mu_{S_2} + p_4\mu_{L_1}\mu_{L_2}) \\ & + \frac{\partial \mu_{L_3}}{\partial a_3}(p_5\mu_{S_1}\mu_{S_2} + p_6\mu_{S_1}\mu_{L_2} + p_7\mu_{L_1}\mu_{S_2} + p_8\mu_{L_1}\mu_{L_2}),\end{aligned} \tag{11}$$

where $\frac{\partial \mu_{S_j}}{\partial a_j} = \mu_{L_j}(1 - \mu_{L_j})/a_j$ and $\frac{\partial \mu_{L_j}}{\partial a_j} = -\frac{\partial \mu_{S_j}}{\partial a_j}$. Note that if an additive updating scheme were used for a_j as suggested in equation (8), the partial derivatives $\frac{\partial \mu_{S_j}}{\partial a_j}$ and $\frac{\partial \mu_{L_j}}{\partial a_j}$ show that larger values of a_j would be updated by smaller amounts and vice versa. This is an undesirable situation in practice, especially considering the effect parameters a_j have on the fuzzy rulebase: a fixed percentage decrease in a_j shifts the intersection point between $\mu_{L_j}(x_j)$ and $\mu_{S_j}(x_j)$ to the left by a fixed increment (see Figure 2), regardless of the value of a_j. Therefore, it is desirable to keep the percentage change in a^j constant (the index j is now used as a superscript so as not to confuse it with the time step), which can be achieved with a *multiplicative* updating scheme. Such a scheme, for the case of TD(0) updating, would first calculate δ_n^j as in equation (8):

$$\delta_n^j = \xi_n^j[c(s_n, \pi(s_n), s_{n+1}) + e^{-\beta\tau_n}\hat{C}(s_{n+1}, p_n) - \hat{C}(s_n, p_n)]\frac{\partial \hat{C}}{\partial a^j}, \tag{12}$$

where ξ_n^j is the learning rate. Then, a^j is updated as follows: $a_{n+1}^j = a_n^j(1 + \xi_n^j\delta_n^j)$ if $\delta_n^j > 0$ and $a_{n+1}^j = a_n^j/(1 - \xi_n^j\delta_n^j)$ otherwise. As one can see, the above expressions imply a nonlinear tuning process for a^j, for which no convergence guarantees have been developed so far. However, the experimental results presented in Section 6 demonstrate that additional tuning of these parameters leads to even better file migration policies.

4.6 A Different View of State Sampling During RL Updates

The standard closed-loop control environment (e.g., the RL problem formulation) assumes that effects of each action can only be observed at the next time step. Thus, the popular RL algorithms have focused on learning the optimal action a for each state s. In some real-world systems, however, the effect of each action can be immediately observed. This is often the case in systems that accept stochastically arriving jobs/requests and make some internal reconfigurations in response (e.g., storage environment considered in this paper, job scheduling environment [16], dynamic resource allocation [17]). In such cases, one can reduce the amount of noise present in RL updates by describing the system's state in such a way so that effects of each action are reflected immediately in the system's state. For example, whenever a decision is made to route a file request from one tier to another, all state variables listed in Section 4.1 are affected for both tiers and immediately reflect the new state of each tier.

The particular RL methodology we propose for tuning the cost-to-go function parameters makes use of the above observation by specifying that the state s_n in equation (8) is the one observed AFTER an upgrading decision (if any) is taken, while the state s_{n+1} is the one right BEFORE the next decision is made. That is, file migration decisions affect only the way the states are sampled for update in equation (8), but the evolution of a state s_n to a state s_{n+1} is always governed by the fixed stochastic behavior of the multi-tier storage environment (which depends on the rates at which file requests are generated, file properties, tier properties, etc.). So far no theoretical convergence analysis has been performed for the proposed sampling approach of choosing at every decision point the smallest-cost state out of those that can be obtained (using the concept described in equation (7) and implemented in the migration criterion (2)), but our positive experimental results suggest its feasibility.

Also note that the above view of the learning process as that of estimating the cost-to-go function for a fixed state transition policy eliminates the need for performing action exploration (which is necessary when we try to learn a new state transition policy), thus making it possible to use this approach online without degrading performance of the currently used policy. For example, if the cost-to-go function for some tier is initialized to prefer some obviously bad states, then after some period of learning the predicted costs of these states will rise to their real levels and the system will start avoiding those states when making scheduling decisions, without an explicit exploration having been performed.

The simulation results in Section 6 demonstrate the power of the RL tuning approach by showing that very good policies can be learned even when the rule parameters p^i are all initialized at 0 (no prior knowledge is used about the relative preferences of various system's states) and the membership function parameters are initialized as described in Section 4.2.

5 Simulation Setup

The experimental results presented in Section 6 are based on a simulated two-tier storage system. Several simplifying assumptions were made about this storage system in order to make a more clear connection between the important system features and the relative performance of various file migration policies.

We assumed 1000 files present in the system, with only one copy of each file being present. Each file stochastically changed its state between being "cold" and being "hot". The probability of a cold file becoming hot after receiving an I/O request was 0.2 and the probability of a hot file becoming cold after receiving an I/O request was 0.005. Requests for a hot file were generated using a Poisson arrival process

with a rate of 0.5, and those for a cold file had an arrival rate of 0.01. The above numbers were chosen so as to result in approximately equal time periods of each file being hot and cold. That is, the average number of accesses a cold file stays cold is $(1-ColdHotSwitchProb)/ColdHotSwitchProb = 4$, and a hot file stays hot is $(1 - HotColdSwitchProb)/HotColdSwitchProb = 199$. The average number of time units a cold file stays cold is then $4/(ColdArrivalRate) = 400$ and a hot file stays hot is $199/(HotArrivalRate) = 398$. Read and write requests were equally probable for each file. At the start of every simulation, each file was initialized to be either hot or cold with probability 0.5. While real hierarchical storage systems will not have Poisson request arrival processes and 2-state Markov switching of files between being "cold" and "hot" (as assumed above), the above model is a reasonable approximation of the salient features of reality, according to domain experts we have consulted.

Many researchers have observed that file sizes in the Internet data traffic have a Pareto distribution $F(x) = 1 - (b/x)^a$ for $x \geq b$ (e.g., [10]). We used this distribution with $a = b = 1$, since these values approximated well several practical workloads we had available. In order to decrease the variance of the experimental results, each simulation started with the same distribution of file sizes, resulting from a particular sampling of the assumed Pareto distribution. The following file sizes were sampled, in descending order of size: 1167, 655, 399, 271, 207, 175, 143, 128, 111, 95, 95. On the other side of the size spectrum, 500 files of size 1 were sampled from the Pareto distribution, 167 files of size 2 (up to file index 667), 83 files of size 3 (up to file index 750), 50 files of size 4 (up to file index 800), 33 files of size 4 (up to file index 833), etc. The storage capacity of the slow tier was chosen to be equal to the total size of all files in the system, 7511. The storage capacity of the fast tier was chosen to be $1/5$ of that, 1502. The bandwidth of the slow tier was 1500, while that of the fast tier was 3000.

The parameter updating rule in equation (8) is executed simultaneously for all tiers whenever each tier in the system has processed k or more requests since the last parameter update. If a very small value of k is used, each tier will not collect enough statistics about response times of requests served since the last update, and the learning will be very noisy. If a very large value of k is used, the parameter updates will be very infrequent, and the learning will be slow. We found experimentally that values between 5 and 50 resulted in statistically the same performance, which is shown in the next section. We have also experimented with different values for the discounting factor $e^{-\beta}$ used in equations (8) and (12), and found that values between 0.9 and 0.99 resulted in statistically the same optimal performance (presented in the next section), which then gradually degraded for values less than 0.9, since they made the RL algorithm more and more short-sighted.

The total training period consisted of 20000 time steps, during which about 20000 parameter updates took place. The training period then followed by a testing period of 20000 time steps, during which the given file migration policy was evaluated. The final results (presented in Table 1) were averaged over 30 trials, making sure that the standard deviation of each performance number is less than 2% of the number itself. When determining the proper schedule for decreasing the learning rate α_n^i in equation (8), we found that the standard approach of setting $\alpha_n^i = 1/n$ decreased α_n^i too fast, so that parameters p^i were not able to reach their final values in 20000 time steps. However, when we used $\alpha_n^i = 0.1/(1 + N \sum_{k=0}^{n-1} \phi^i(s_k))$ with $50 < N < 500$, the magnitudes of updates were large enough for p^i to reach the neighborhood of their final values.

The experimental results showed a slightly better performance when the learning rate ξ_n^j used for updating the membership function parameters a^j in equation (12) was kept constant rather than being decreased after every update. Note that the magnitude of δ_n^j in equation (12) depends on the magnitude of $\frac{\partial \hat{C}}{\partial a^j}$, which in turn depends on the relative values of parameters p^i. Therefore, in order to have all

parameters a_j updated at approximately the same constant rate, the expected value of $\xi_n^j \delta_n^j$ should be equal to the same small constant for all j. In order to achieve that, we first estimated for each j the average value $\bar{\delta}_n^j$ of $|\delta_n^j|$ during the first 5000 updates of parameters p^i (while a_j were not updated). We then found that setting $\xi_n^j = 0.001/\bar{\delta}_n^j$ resulted in reasonably sized updates.

When using an FRB with membership functions $\mu_{L_j}(x_j) = 1/(1+a_j e^{-b_j x})$ and $\mu_{S_j}(x_j) = 1-\mu_{L_j}(x_j)$ (as shown in Figure 2) to approximate some cost function, it is important to choose the initial values of parameters a_j and b_j appropriately. Assuming that a_j are initially chosen so that $\mu_{L_j}(x_j)$ and $\mu_{S_j}(x_j)$ intersect at the mean value of x_j, if b_j is chosen too large, then μ_{L_j} and μ_{S_j} will look like step functions, while if b_j is chosen too small, then μ_{L_j} and μ_{S_j} will look like two linear functions over the range of x_j. As the practice shows, the maximal approximation capability of an FRB lies somewhere between the two extremes described above.

In order to choose the appropriate value for b_j, one has to first estimate the range of the values of x_j. We achieve that by observing the storage system for the first 5000 time steps (making about 5000 file upgrading decisions) and computing the mean $\bar{x}_j$ and the standard deviation σ_j for each variable x_j. The expected range for x_j (the one specified as [0,1] in Figure 2) is then set to be $[\bar{x}_j - 2 \cdot \sigma_j, \bar{x}_j + 2 \cdot \sigma_j]$. Let $minX_j = \bar{x}_j - 2 \cdot \sigma_j$, $maxX_j = \bar{x}_j + 2 \cdot \sigma_j$, $L_j = maxX_j - minX_j$, $C_j = (maxX_j + minX_j)/2$. We found that the value $b_j = 7.33/L_j$ as shown in Figure 2 resulted in the best approximations of the tier cost functions, which degraded gradually as b_j was increased or decreased. The value of a_j was then set to $e^{b_j C_j}$, so that $\mu_L(C_j) = \mu_S(C_j) = 0.5$. Note that given our choice for b_j, all parameters a_j were set to the same initial value of $e^{0.5 \cdot 7.33}$. The parameters p_i were all initialized to 0.

6 Simulation Results

Table 1 shows the average weighted response time (AWRT) of the various policies considered. The first set of policies address the case when no file migration was performed and the policies differ only in the way the files were initially allocated in the system. In one case, each file, starting from the smallest one, was randomly assigned to one of the two tiers with equal probability if both tiers had enough space for it; otherwise, the file was written to the slow tier. In the other case, the smallest files were written to the fast tier until it became full (fitting 841 smallest files), at which point all remaining files were written to the slow tier. Note that AWRT for the second policy is 20 times smaller than that for the first policy! Such a large difference can be explained by the fact that it is very beneficial to keep all small files together and all large files together. Otherwise, small files can get stuck in the queue behind large files and experience a very large AWRT. All subsequent policies were initialized with the smallest-first allocation.

The second set of results addresses the case of pre-specified file migration policies. In all of the cases file migration was found to give better results if files were upgraded only following a write request, which would simply be re-routed to the next fastest tier. That is, the slow tier most of the time did not experience any change in its load, since it would only have to execute the write requests for the files that were swapped out from the fast tier, which were together on average of the same size as the file being upgraded. However, when migration following a read request was enabled, the slow tier would actually experience an increase in its load, since it would have to first respond to the read request and then execute the write request for the files swapped out from the fast tier.

The first two policies in this group address the case when a write request was routed from the slow tier to the fast tier if the file "temperature", computed according to equation (1), was higher than that of

Policy	Cost (AWRT)
No Migration, random allocation	0.096
No Migration, smallest-first allocation	0.0043
Least Recently Used Replacement 1	0.085
Least Recently Used Replacement 2	0.027
Size-Temperature Replacement 1	0.0027
Size-Temperature Replacement 2	0.0026
RL Tuning of p^i	0.0018
RL Tuning of p^i and a^j	0.0017

Table 1. Average Weighted Response Time (AWRT) of various policies

the "coldest" file in the fast tier. If the fast tier did not have enough space to accept the file, some files were swapped out to the slow tier, in the order of being least recently used (LRU). This file replacement policy is currently being used in most commercial storage products and also in managing CPU cache space. The first policy allowed all files to be upgraded, while the second policy allowed only files of size less than 200 to be upgraded. The second policy achieved a 3 time smaller AWRT because, once again, it prevented very large files from being in the same tier as most of the small files. However, it still allowed for a larger than desired mixing of large and small files, and consequently its AWRT was still 5 times larger than that of the no-migration policy with the smallest-first file allocation. Both of these policies (and all file migration policies discussed below) were given 5000 time steps at the beginning of the simulation to settle into a steady-state behavior, before the testing period (or the training period in the case of RL-based policies) began.

The next two policies in this group used a replacement policy that considered both the temperature and the size of the files. This replacement policy would first try to remove a file that is of the same size or bigger and is also colder than the file being upgraded. If no such file was found, then the policy would try to remove at most two colder files that are of the same size or smaller, searching through all files in the order of decreasing size. If enough space could be created, the file was upgraded; otherwise, no action would be taken. That is, this replacement policy tried to simultaneously achieve the objectives of keeping hottest AND smallest files in the fast tier. The difference between these two policies is that the first one used the same upgrading criterion as before, while the second one used a stricter criterion of upgrading only those files that are hotter than the coldest file in the fast tier AND hotter than the average file temperature in the slow tier. As one can see, AWRT of these two policies is 10 times smaller than that of the policies based on LRU replacement.

The final set of policies are those where migration was performed based on minimizing the tier cost functions, which were previously tuned with RL. More specifically, a write request would be forwarded to the fast tier if the migration criterion (2) was satisfied. The parameters were tuned using the TD(λ) RL algorithm presented in equations (8) and (12) using the whole range of possible values of λ. Interestingly, the difference in results between the various values of λ was statistically insignificant. Both policies allowed tuning the p^i parameters, but only the second policy allowed a simultaneous tuning of the a_j parameters as described in Section 5. We found that the values of parameters a_j were changed by at most

50% regardless of the length of the training period. Recalling that parameters b_j were set to $7.33/L_j$, a 50% change in a_j implies a shift of the intersection point between the membership functions μ_{L_j} and μ_{S_j} by about 5% of the range of the variable x_j. A greater change was not observed possibly because the initial statistical estimate of this range (described in Section 5) was sufficiently accurate. That is, when statistical range estimation is used, the parameters a_j do not really need to be tuned, and hence the results in Table 1 show only a minor performance improvement due to tuning a_j.

The parameter tuning process was based on a Policy Iteration (PI) approach, developed by Ronald Howard [3], which was previously mentioned when presenting equation (6). That is, during the first iteration the Size-Temperature Replacement 1 policy would be executed for 20000 time steps, and the tier cost functions C_j^0 would be learned for this policy for tiers $j = 1$ and 2. During the second iteration, migration would be performed based on inequality (2) with the cost functions C_j^0, and the new cost functions C_j^1 would be learned for this policy for 20000 time steps. During the third iteration, migration would be performed based on inequality (2) with the cost functions C_j^1, and the new cost functions C_j^2 would be learned for this policy for 20000 time steps. It was found that statistically significant performance improvement was achieved only during the first three iterations of the PI cycle, reducing AWRT by 35% in comparison to the initial policy. This corresponds to the practical observations made by many researchers that the PI cycle can learn a very good policy after only several iterations. Another major practical benefit of the PI approach is that it never uses a "bad policy: each subsequent policy is at least as good as the previous one, making PI a good candidate for on-line learning.

We are not aware of any successful attempts to tune the basis function parameters using RL, and some researchers in fact have reported that such a tuning decreases performance (e.g., [4, 15]). Therefore, the procedure presented in this paper for tuning a_j is an important contribution to the practice of combining RL with function approximation architectures consisting of linear combinations of basis functions.

7 Conclusion

This paper presented a novel framework for dynamically re-distributing files in a multi-tier storage system, which explicitly minimizes the expected future average request response time. The previous approaches to storage system management are based on fixed heuristic policies and do not perform any performance optimization. The proposed framework uses a Reinforcement Learning (RL) methodology for tuning parameters of tier cost functions based on which file migration is performed. Any parameterized function approximation architecture can be used in the proposed framework to represent tier cost functions. A fuzzy rulebase architecture was chosen here for its ease of management. The RL methodology was applied to tuning the parameters of the fuzzy rulebase, which resulted in a 35% performance improvement relative to the best found pre-specified file migration policy. The experimental results were robust with respect to significant perturbations of the key parameters.

8 Acknowledgment

The author would like to thank Harriet Coverston from Sun Microsystems for the helpful information about hierarchical storage systems, as well as Declan Murphy and Victoria Livschitz for the help with presenting this material.

References

[1] R. Das, G. J. Tesauro, W. E. Walsh. "Model-Based and Model-Free Approaches to Autonomic Resource Allocation," IBM Technical Report RC23802, 2005.

[2] S. W. Hasinoff. "Reinforcement Learning for Problems with Hidden State," Technical Report, University of Toronto, Department of Computer Science, 2002.

[3] R. A. Howard. *Dynamic Programming and Markov Processes*. John Wiley & Sons, Inc., New York, 1960.

[4] R. M. Kretchmar and C. W. Anderson. "Comparison of CMACs and RBFs for Local Function Approximators in Reinforcement Learning." In Proceedings of the IEEE International Conference on Machine Learning, pp. 834-837, Houston, TX, 1997.

[5] L. J. Lin and T. M. Mitchell."Memory Approaches to Reinforcement Learning in Non-Markovian Domain," Carnegie Mellon School of Computer Science Technical Report CMU-CS-92-138, 1992.

[6] C. Lu, G. A. Alvarez, and J. Wilkes. "Aqueduct: online data migration with performance guarantees," Conference on File and Storage Technology (FAST'02), pp. 219 - 230, Monterey, CA, 2002. Published by USENIX, Berkeley, CA.

[7] A. K. McCallum. *Reinforcement Learning with Selective Perception and Hidden State*, PhD. Thesis, University of Rochester, Department of Computer Science, 1995.

[8] J. Menon, D. A. Pease, B. Rees, L. M. Duyanovich, and B. L. Hillsber. "IBM Storage Tank – A Heterogeneous Scalable SAN File System." *IBM Systems Journal*, Vol. 42, No. 2, pp. 250 - 267, 2003.

[9] N. Meuleau, L. Peshkin, K.-E. Kim, and L. P. Kaelbling. "Learning finite-state controllers for partially observable environments," in Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence (UAI), pp. 427 – 436, 1999.

[10] V. Paxson and S. Floyd. "Wide-Area Traffic: The Failure of Poisson Modeling." *IEEE/ACM Transactions on Networking*, Vol. 3, No. 3, pp. 226 - 244, 1995.

[11] M. Sinnwell and G. Weikum, "A Cost-Model-Based Online Method for Distributed Caching." In Proceedings of the Thirteenth International Conference on Data Engineering (ICDE), pp. 532 - 541, Birmingham, UK, 1997.

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

[13] Sun Microsystems Inc. "Sun StorEdge QFS and SAM-FS Software," Technical white paper. Available electronically at http://www.sun.com/storage/white-papers/qfs-samfs.pdf, 2004.

[14] 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, pp. 674 – 690, 1997.

[15] P. Vamplew and R. Ollington. “Global Versus Local Constructive Function Approximation for On-Line Reinforcement Learning.” Presented at the 18th Australian Joint Conference on Artificial Intelligence, December 5-9, Sydney, Australia, 2005. Published in Springer-Verlag Lecture Notes in Computer Science, Vol. 3809, pp. 113-121, 2005.

[16] D. Vengerov. “Reinforcement Learning Framework for Utility-Based Scheduling in Resource-Constrained Systems.” Sun Microsystems Laboratories Technical Report TR-2005-141, Feb. 1, 2005.

[17] D. Vengerov. “A Reinforcement Learning Approach to Dynamic Resource Allocation.” Sun Microsystems Laboratories Technical Report TR-2005-148, Sep. 1, 2005.

[18] D. Vengerov. “Dynamic Tuning of Online Data Migration Policies in Hierarchical Storage Systems using Reinforcement Learning.” Sun Microsystems Laboratories Technical Report TR-2006-157, June 19, 2006.

[19] A. Verma, U. Sharma, J. Rubas, D. Pease, M. Kaplan, R. Jain, M. Devarakonda and M. Beigi, “An Architecture for Lifecycle Management in Very Large File Systems,” In Proceedings of the 22nd IEEE / 13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST), pp. 160 - 168, 2005.

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